15. October 2010

Handling Large RDF Graphs with MapReduce

Semantic Rhine

Martin Przyjaciel-Zablocki
Alexander Schätzle

University of Freiburg
Databases & Information Systems Group

Overview

1. Motivation

- Analysis of large RDF Graphs

Large RDF Graphs

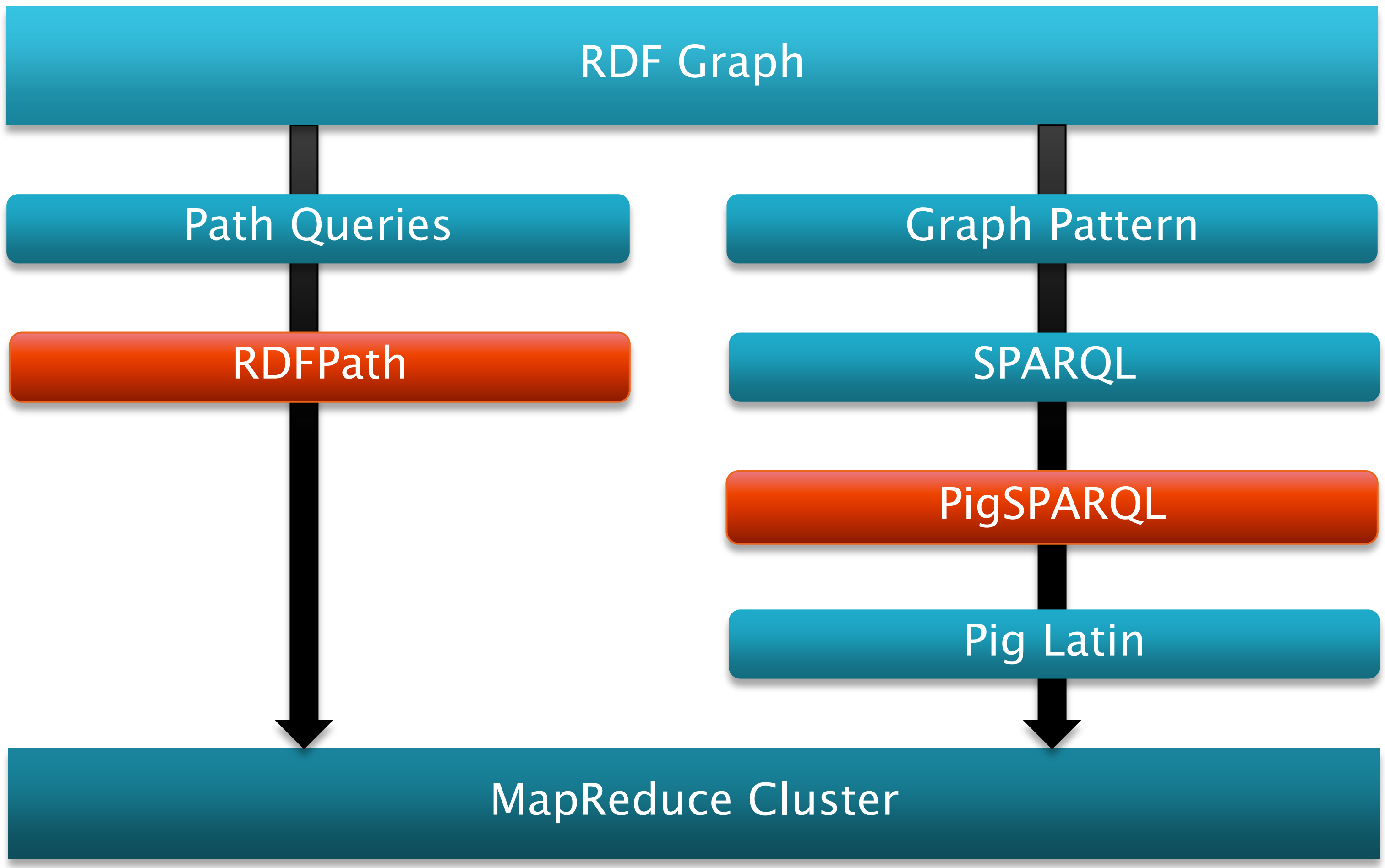
- **Facebook** (2010)[1]
  - > 500 million active users
  - > 900 million interactive objects (sites, groups, events, …)
  - Usage: > 700 billion minutes per month
  - Can be expressed as RDF Graphs

- **How to handle such large RDF Graphs?**

- **Approach:** **Distributed analysis of large RDF Graphs using MapReduce**

Source:
(1) Facebook Press Room (12.10.2010)
http://www.facebook.com/press/info.php?statistics

Two query languages for analysis



2. MapReduce

» Principles & Basic Concepts

MapReduce

- **Google's MapReduce**
  - Automatic parallelization of computations
  - Fix and simple level of abstraction: Map & Reduce
- **Distributed File System**
  - Clusters of commodity hardware
    → Fault tolerance by replication
  - Very large files / write-once, read-many-times
- **Hadoop**
  - Open Source implementation (Apache project)
  - Used by Yahoo, Facebook, Amazon, IBM, Last.fm, …
  - more

3. RDFPath

- Path queries on large RDF Graphs
  Martin Przyjaciel-Zablocki

RDFPath

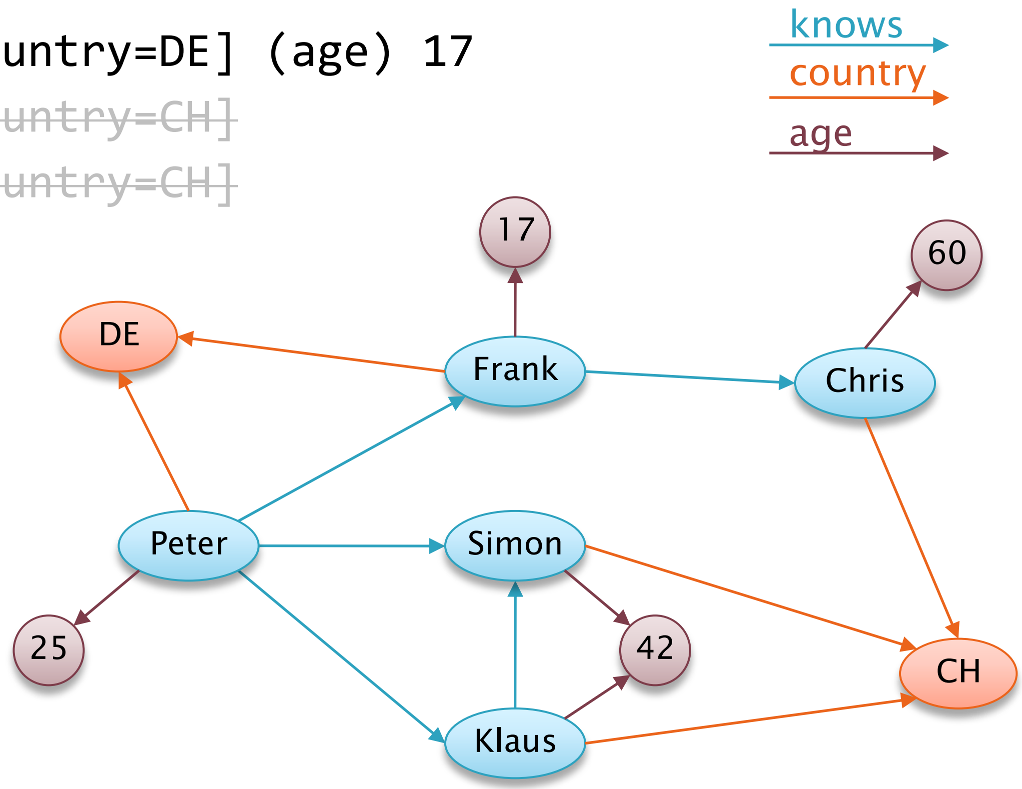
- **Requirements**
  - Navigational queries over RDF Graphs
  - Extendibility
  - Particularly with regard to a MapReduce evaluation

- **Idea**
  - Declarative path specification with XPath like location steps
  - Every location step can be mapped to one MapReduce job

Example (1)

- Peter :: knows[country=equals(DE)] > age.

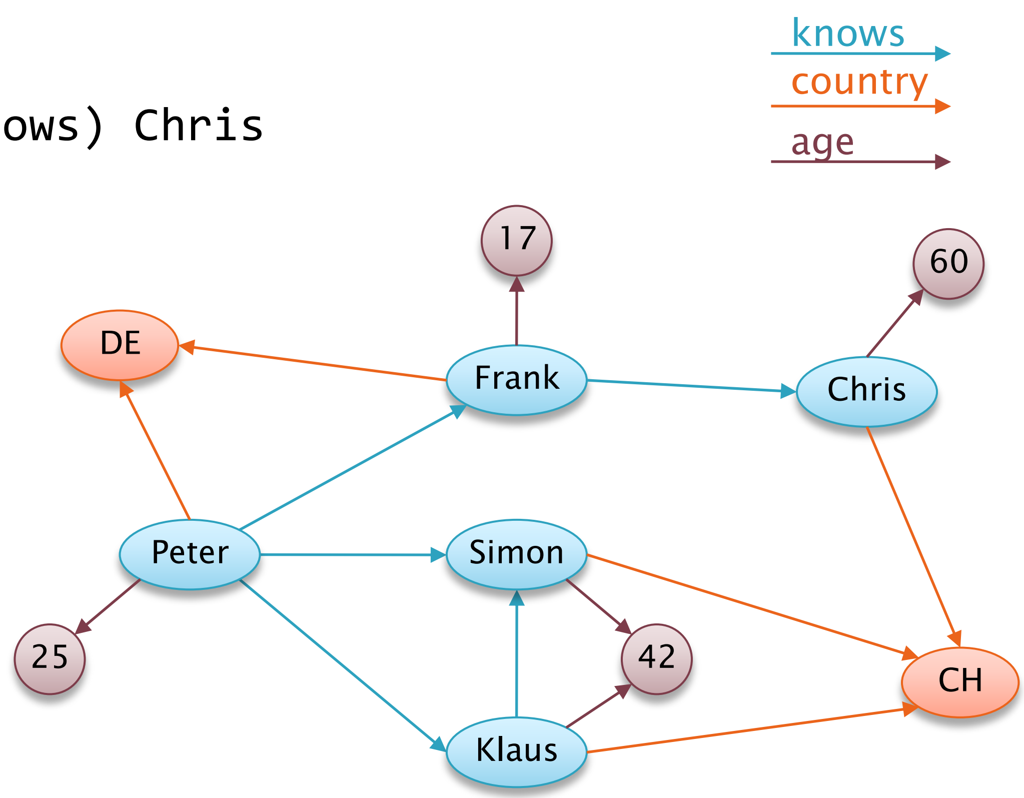
- Results
  - Peter (knows) Frank [country=DE] (age) 17
  - ~~Peter (knows) Klaus [country=CH]~~
  - ~~Peter (knows) Simon [country=CH]~~



Example (2)

- Peter :: knows(*3).

- Results
  - Peter (knows) Frank
  - Peter (knows) Frank (knows) Chris
  - Peter (knows) Klaus
  - Peter (knows) Simon



more

Supported features

- Starting nodes
  - fixed or arbitrary
- Location step follows edge
- Filters & sub queries
- Shortest path queries
- Avoidance of cycles
- Different types of result
  - paths, nodes, aggregations,…

Further components

- RDFPath-Store
  - Build on the top of HDFS + local storage
  - Vertical partitioning related to predicates (edges)
  - Optional Dictionary Encoding

- Query-Engine

```
Peter :: knows(*2)  >  knows.
```

previous paths

```
Peter (knows) Frank (knows) Chris
Peter (knows) Simon
Peter (knows) Klaus
…
```

Join

knows

Chris	Peter
Johan	Frank
Frank	Chris
…	

system

rsj

Evaluation

- Hadoop cluster with 10 servers
- Real Last.fm & generated SP^{2}Bench datasets

- **Results**
  - Promising scaling behavior
  - Evaluated up to 1.6 billion triples
  - Considered problems:
    Shortest path, Erdoes-number, Six-degrees of sep., …
  - Dictionary Encoding reduces data but with significant Dictionary lookup costs

4. PigSPARQL

Translating SPARQL to Pig Latin
Alexander Schätzle

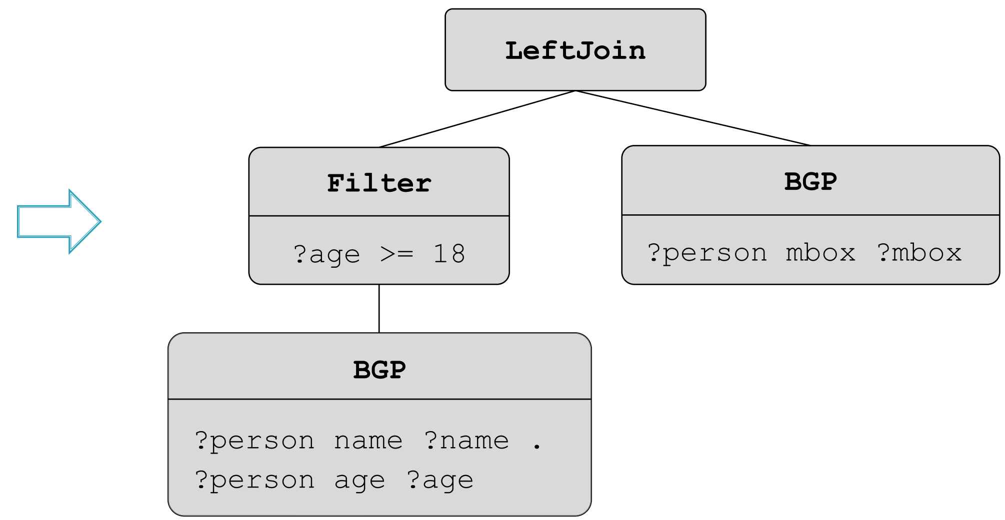
Pig Latin

- **Advantages of MapReduce**
  - Parallelization done by the system
  - Good fault tolerance & scalability
- **Drawbacks of MapReduce**
  - „Low-Level“ to implement & hard to maintain
  - No primitives like JOIN or GROUP
- **Pig Latin**
  - „High-Level“ language for data analysis with Hadoop
  - Link between user & MapReduce
  - Automatic translation into MapReduce jobs
  - more

Translation of SPARQL (1)

- **1. Step**
  - Convert SPARQL Query into SPARQL Algebra-Tree

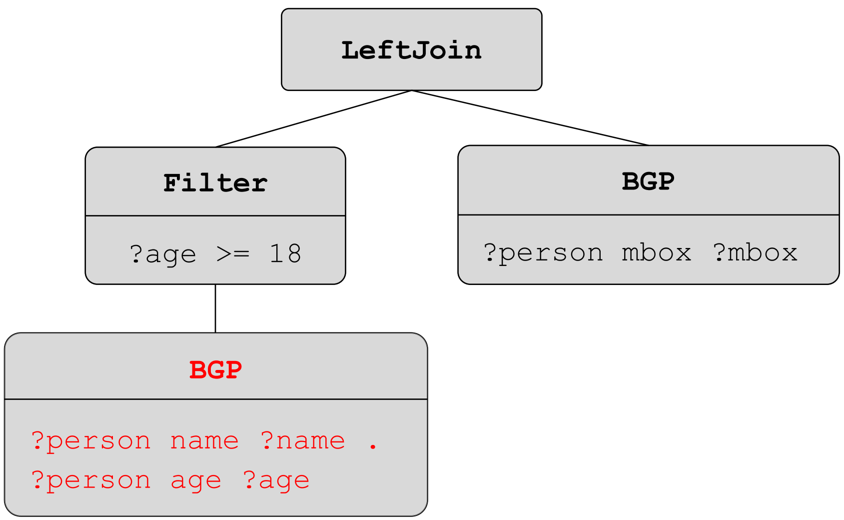
```
SELECT *
WHERE {
  ?person foaf:name ?name.
  ?person foaf:age ?age.
  FILTER (?age >= 18)
  OPTIONAL {
    ?person foaf:mbox ?mbox
  }
}
```



Translation of SPARQL (2)

2. Step

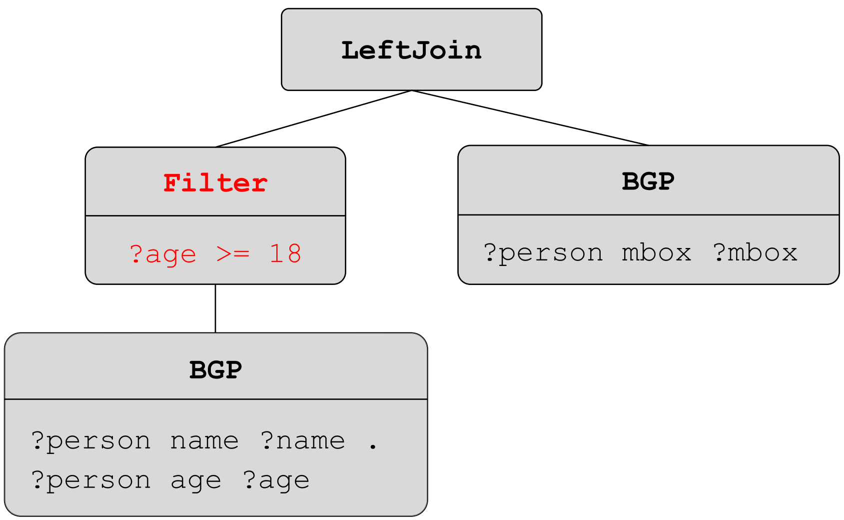
- Translate Algebra-Tree into Pig Latin Program




```
indata = LOAD 'pathToFile' USING myLoad() AS (s,p,o);

f1 = FILTER indata BY p=='foaf:name';
t1 = FOREACH f1 GENERATE s AS person, o AS name;
f2 = FILTER indata BY p=='foaf:age';
t2 = FOREACH f2 GENERATE s AS person, o AS age;
j1 = JOIN t1 BY person, t2 BY person;
BGP1 = FOREACH j1 GENERATE t1::person AS person,
       t1::name AS name, t2::age AS age;
```

Translation of SPARQL (2)

- **2. Step**
  - Translate Algebra-Tree into Pig Latin Program

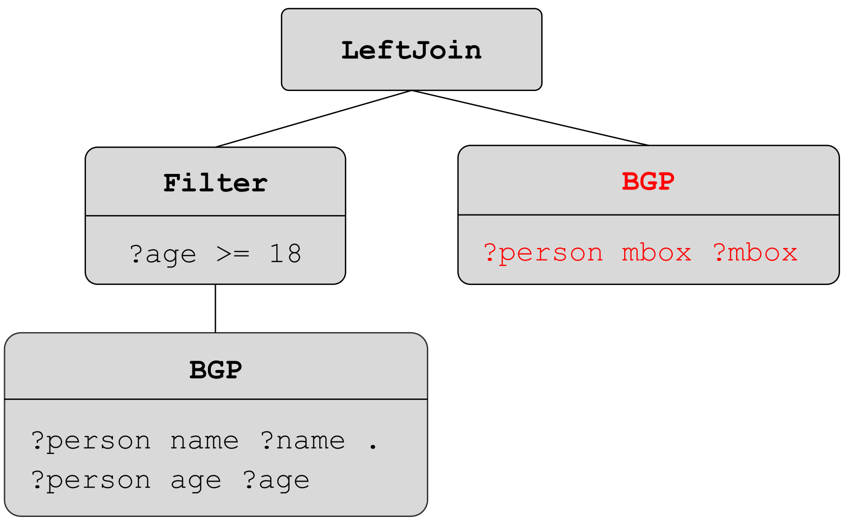



```
indata = LOAD 'pathToFile' USING myLoad() AS (s,p,o);

f1 = FILTER indata BY p=='foaf:name';
t1 = FOREACH f1 GENERATE s AS person, o AS name;
f2 = FILTER indata BY p=='foaf:age';
t2 = FOREACH f2 GENERATE s AS person, o AS age;
j1 = JOIN t1 BY person, t2 BY person;
BGP1 = FOREACH j1 GENERATE t1::person AS person,
       t1::name AS name, t2::age AS age;

F1 = FILTER BGP1 BY age >= 18;
```

Translation of SPARQL (2)

2. Step

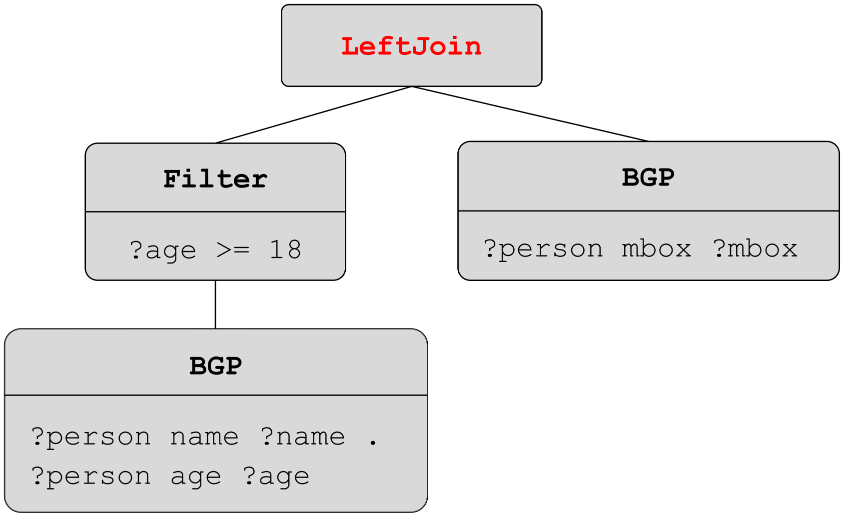
- Translate Algebra-Tree into Pig Latin Program




```
indata = LOAD 'pathToFile' USING myLoad() AS (s,p,o);

f1 = FILTER indata BY p=='foaf:name';
t1 = FOREACH f1 GENERATE s AS person, o AS name;
f2 = FILTER indata BY p=='foaf:age';
t2 = FOREACH f2 GENERATE s AS person, o AS age;
j1 = JOIN t1 BY person, t2 BY person;
BGP1 = FOREACH j1 GENERATE t1::person AS person,
       t1::name AS name, t2::age AS age;

F1 = FILTER BGP1 BY age >= 18;

f1 = FILTER indata BY p=='foaf:mbox';
BGP2 = FOREACH indata GENERATE s AS person, o AS mbox;
```

Translation of SPARQL (2)

- 2. Step
  - Translate Algebra-Tree into Pig Latin Program




```
indata = LOAD 'pathToInput' USING myLoad() AS (s,p,o);

f1 = FILTER indata BY p=='foaf:name';
t1 = FOREACH f1 GENERATE s AS person, o AS name;
f2 = FILTER indata BY p=='foaf:age';
t2 = FOREACH f2 GENERATE s AS person, o AS age;
j1 = JOIN t1 BY person, t2 BY person;
BGP1 = FOREACH j1 GENERATE t1::person AS person,
       t1::name AS name, t2::age AS age;

F1 = FILTER BGP1 BY age >= 18;

f1 = FILTER indata BY p=='foaf:mbox';
BGP2 = FOREACH indata GENERATE s AS person, o AS mbox;

lj = JOIN F1 BY person LEFT OUTER, BGP2 BY person;
LJ1 = FOREACH lj GENERATE F1::person AS person,
      F1::name AS name, F1::age AS age,
      BGP2::mbox AS mbox;

STORE LJ1 INTO 'pathToOutput' USING myStore();
```

Optimizations

Three Levels of Optimization:

- ## SPARQL Algebra
  - Filter Optimizations (Pushing, Splitting, Substitution)
  - Triple-Pattern Reordering by Selectivity

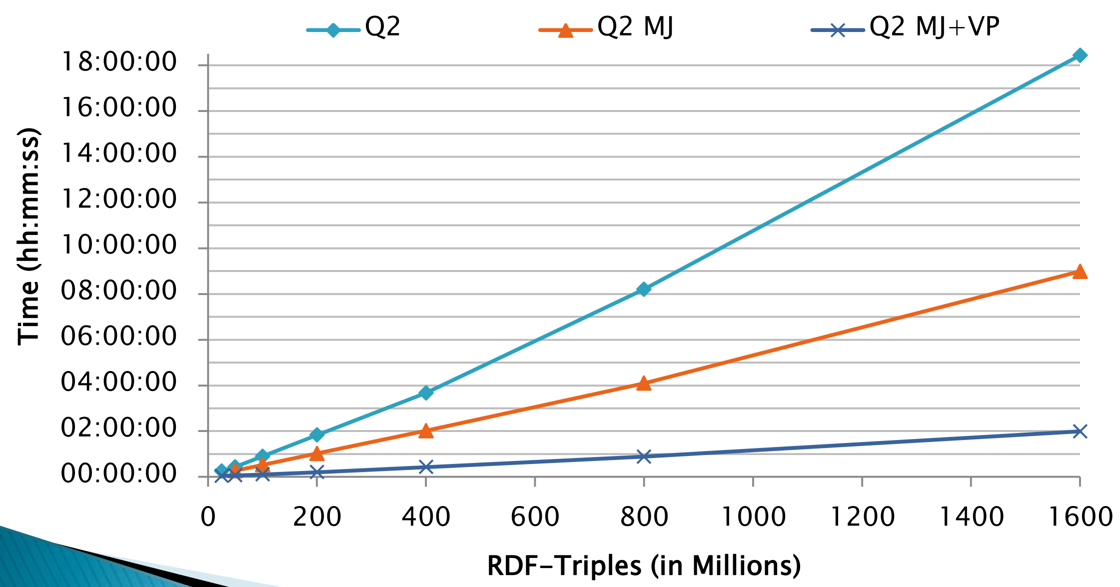
- ## Algebra Translation
  - Delete unnecessary Data as early as possible
  - Multi-Joins to reduce the Number of Joins

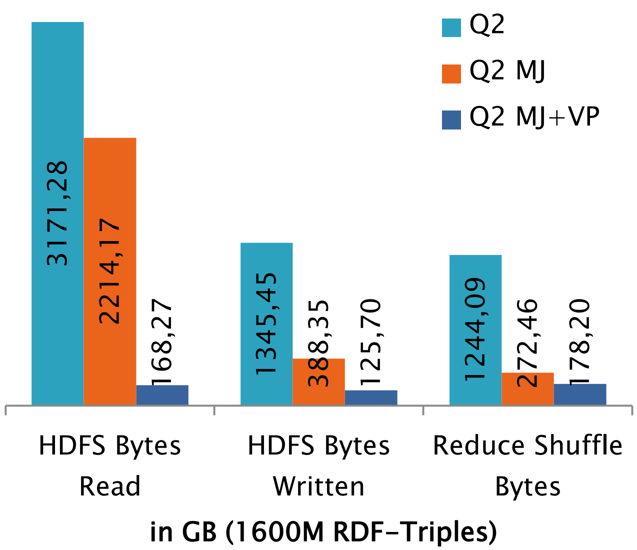
- ## Data Representation
  - Vertical Partitioning of the RDF-Data by Predicate

Evaluation

- Native Translation needs 8 Joins + 1 Outer Join
- Multi-Join reduces the number of Joins
- Vertical Partitioning reduces the Input-Data

```
SELECT ?inproc ?author ?booktitle ?title
       ?proc ?ee ?page ?url ?yr ?abstract
WHERE {
  ?inproc rdf:type bench:Inproceedings .
  ?inproc dc:creator ?author .
  ?inproc bench:booktitle ?booktitle .
  ?inproc dc:title ?title .
  ?inproc dcterms:partOf ?proc .
  ?inproc rdfs:seeAlso ?ee .
  ?inproc swrc:pages ?page .
  ?inproc foaf:homepage ?url .
  ?inproc dcterms:issued ?yr
  OPTIONAL {
    ?inproc bench:abstract ?abstract
  }
}
ORDER BY ?yr
```





5. Summary

- Handling Large RDF Graphs with RDFPath & PigSPARQL on MapReduce

Summary

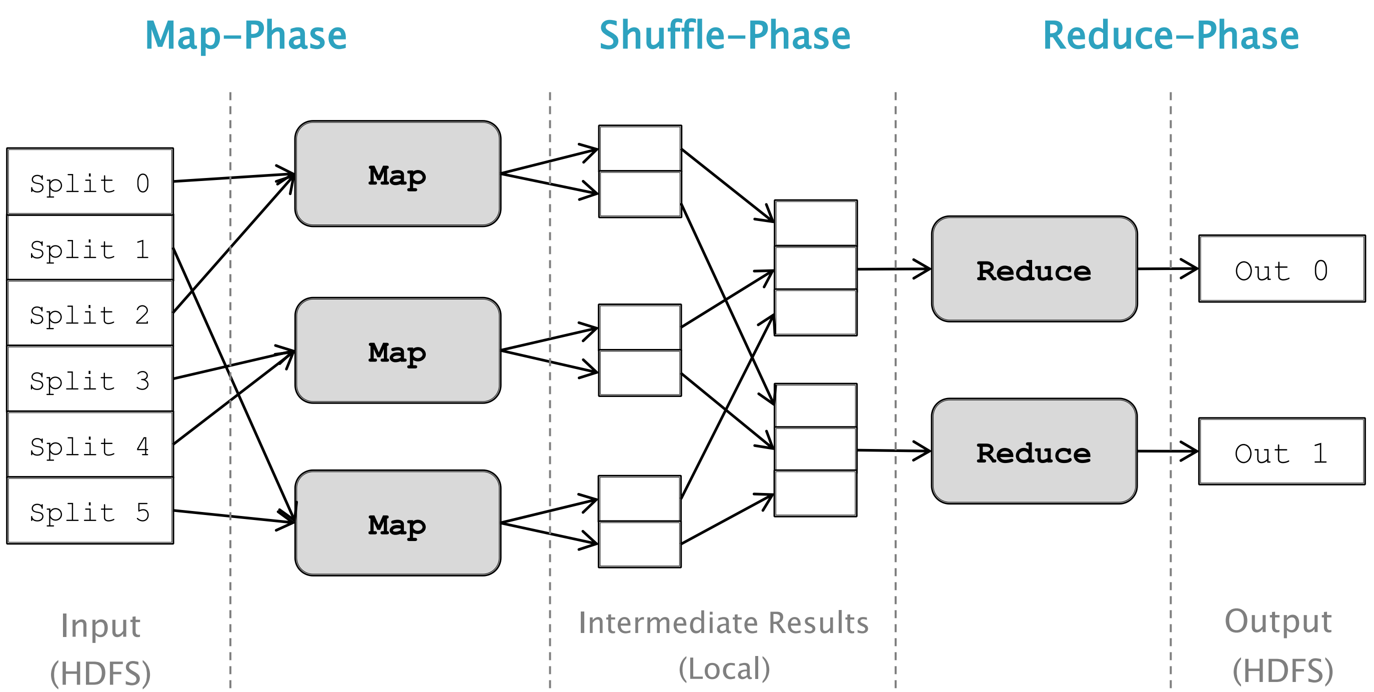
- RDFPath is especially suited for the execution of path queries on large RDF Graphs with MapReduce
- PigSPARQL allows the efficient execution of SPARQL queries with MapReduce
- Handling up to 1.6 Billion RDF Triples
- Both approaches show a promising scaling behavior
- I/O is the dominating bottleneck
  → Optimization means reducing the I/O

Thanks for your attention.

Backup Slides

MapReduce (2)

- **Steps of a MapReduce execution**



MapReduce (3)

- Signature of a Map-Function
  - `map(in_key, in_value) -> (out_key, intermediate_value) list`
- Signature of a Reduce-Function
  - `reduce(out_key, intermediate_value list) -> out_value list`
- back

Pig Latin – Data Model

- Flexible, nested Data Model
- 4 Datatypes:

Atom:	'Bob'
Tuple:	('John', 'Doe')
Bag:	{ ('Bob' , 'Sarah') ('Peter', ('likes', 'football')) }
Map:	['knows' -> {('Sarah')} 'age' -> 24]

- Tupelwise Loading of Data with "User Defined Function"
- every Field of a Tuple can have a Name and a Datantype

Pig Latin – Operators (1)

FOREACH: Apply Processing on every Tuple

Ex: result = FOREACH input GENERATE field1*field2 AS mul ;

input

field1	field2
2	3
4	7

⇨

result

mul
6
28

FILTER: Delete unwanted Tuples

Ex: adults = FILTER persons BY age >= 18 ;

persons

name	age
Bob	21
Sarah	17

⇨

adults

name	age
Bob	21

Pig Latin – Operators (2)

[OUTER] JOIN: Combine two or more Relations

Ex: result = JOIN left BY field1 [LEFT OUTER], right BY field2 ;

left

field1
a
b

⋈

right

field1	field2
4	a
7	a

⇨

result

left::field1	right::field1	right::field2
a	4	a
a	7	a

left

field1
a
b

⋈ Outer

right

field1	field2
4	a
7	a

⇨

result

left::field1	right::field1	right::field2
a	4	a
a	7	a
b		

Pig Latin – Operators (3)

UNION: Ex: result = UNION rel1, rel2 ;

rel1

field1	field2
a	1

U

rel2

field1	field2
b	3

⇨

result

field1	field2
a	1
b	3

ORDER: Ex: result = ORDER input BY field1 ;

input

field1	field2
3	a
1	b

⇨

result

field1	field2
1	b
3	a

back

Last.fm Example

- Michael_Jackson :: artistTracks
  [trackAlbum = equals(Michael_Jackson_-_Thriller)]
  > trackSimilar [trackDuration = min(50000)]
  > trackTopFans [userCountry = equals(DE)].

- Results
  - Michael_Jackson (artistTracks)
    Michael_Jackson_-_Beat_It (trackSimilar)
    Michael_Jackson_-_Billie_Jean (trackTopFans) Mark
  - Michael_Jackson (artistTracks)
    Michael_Jackson_-_Someone_in_the_Dark (trackSimilar)
    Rihanna_-_Russian_Roulette (trackTopFans) Megan

back

Reduce-Side Join

- Example: `* :: knows(*2)` > `knows.`

previous paths

```
Peter (knows) Frank (knows) Chris
Peter (knows) Simon (knows) Johan
Klaus (knows) Simon (knows) Johan
Frank (knows) Chris
Peter (knows) Klaus
              …
```

Reduce-Side Join

knows

```
Chris   Peter
Johan   Frank
Johan   Lukas
Frank   Chris
    …
```

Reduce-Side Join (2)

- Mapper Input — Mapper Output

previous paths

```
Peter (knows) Frank (knows) Chris
Peter (knows) Simon (knows) Johan
Klaus (knows) Simon (knows) Johan
Frank (knows) Chris
Peter (knows) Klaus
…
```

knows

```
Chris   Peter
Johan   Frank
Johan   Lukas
Frank   Chris
…
```

...

...

Key	Value
(Chris, 1)	Peter (knows) Frank (knows) Chris
(Johan, 1)	Peter (knows) Simon (knows) Johan
(Johan, 1)	Klaus (knows) Simon (knows) Johan
(Chris, 1)	Frank (knows) Chris
(Klaus, 1)	Peter (knows) Klaus
…	
(Chris, 0)	Peter
(Johan, 0)	Frank
(Johan, 0)	Lukas
(Frank, 0)	Chris
…	

Reduce-Side Join (3)

- **Reducer's strategy (sorting phase):**
  (1) Partition according to the first keypair % #reducer
  (2) Sort within a partiton according the whole keypair

- **Consequences**
  - A Reducer gets all „values" with the same first keypair
  - The „values" within a partiton contains at first all new nodes and thereafter all previous paths

Reduce-Side Join (4)

Reducer Input

```
                             Chris
Peter
Manu
Peter (knows) Frank (knows) Chris
Frank (knows) Chris
                 …
```

```
                             Johan
Frank
Lukas
Peter (knows) Simon (knows) Johan
Klaus (knows) Simon (knows) Johan
                 …
```

Reducer Output

```
~~Peter (knows) Frank (knows) Chris (knows) Peter~~
Peter (knows) Frank (knows) Chris (knows) Manu
Frank (knows) Chris (knows) Peter
Frank (knows) Chris (knows) Manu
                   …

Peter (knows) Simon (knows) Johan (knows) Frank
Peter (knows) Simon (knows) Johan (knows) Lukas
Klaus (knows) Simon (knows) Johan (knows) Frank
Klaus (knows) Simon (knows) Johan (knows) Lukas
```

back

RDFPath System

RDFPath Query

Query Engine

Intermediate Language	Sequence of MapReduce Jobs

Evaluation

MapReduce-Framework	HDFS	RDFPath-Store

back