Linked Data Management on the Cloud

Zoi Kaoudi
Inria Saclay, France

(Extended version of ICDE 2013 tutorial with Ioana Manolescu)

Open Data tutorials @ WOD'2013

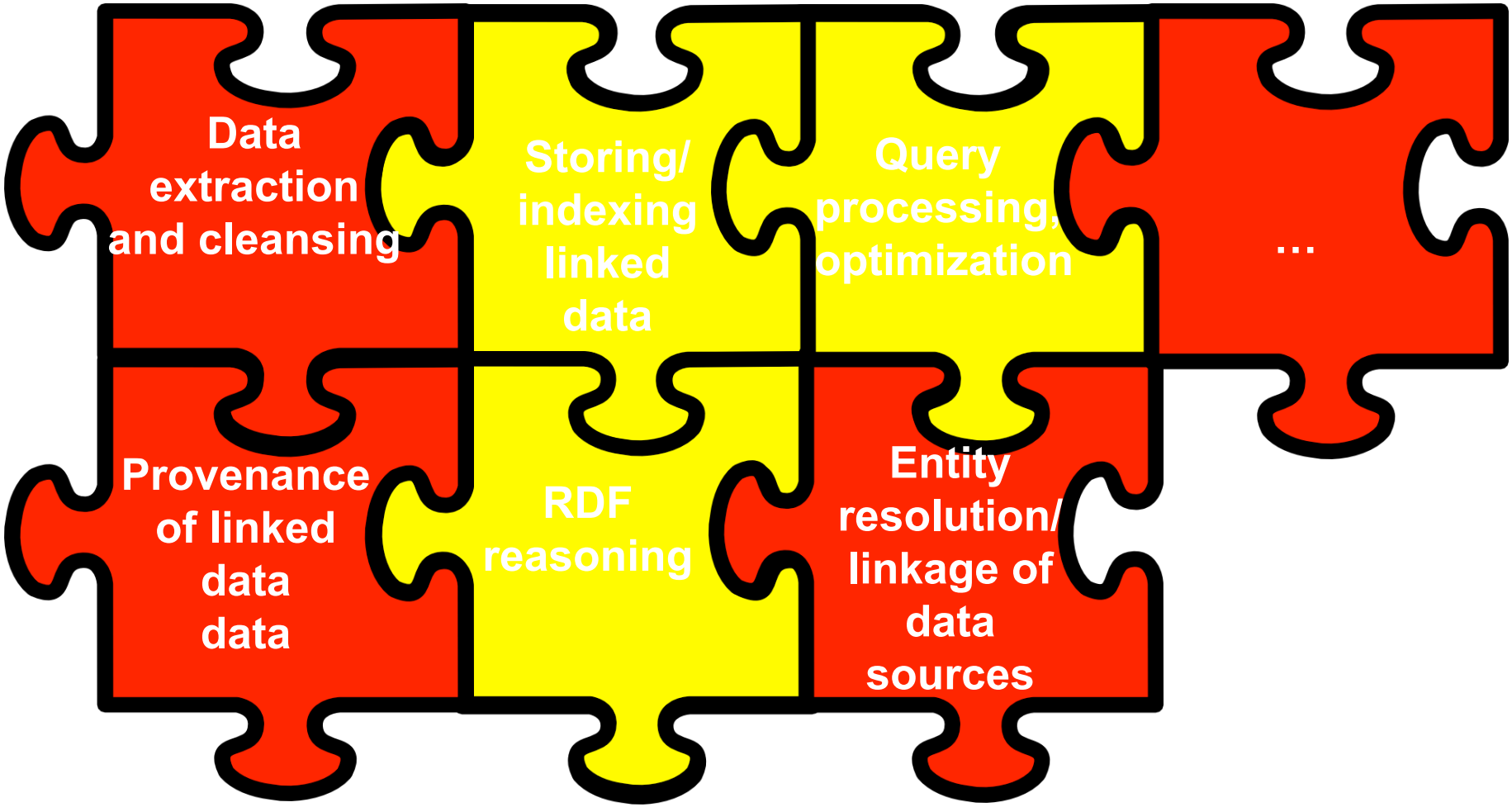
Linked [Open] Data

- Tim Berners-Lee in TED 2009 conference

  "Linked Data is the Semantic Web done right"

- Linked Data principles:
  - Use URIs to represent data
  - Use HTTP URIs (make data dereferenceable)
  - Give information for a piece of data using W3C standards
  - Include links to other URIs
- Linked Data Management → RDF data management

Linked data management tasks

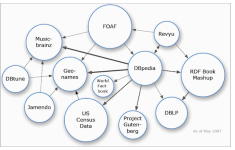


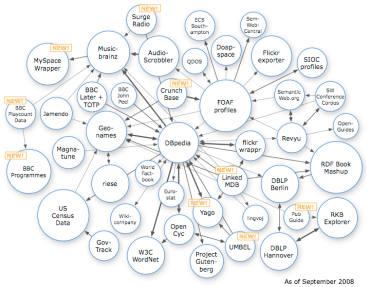
Where is RDF found today?

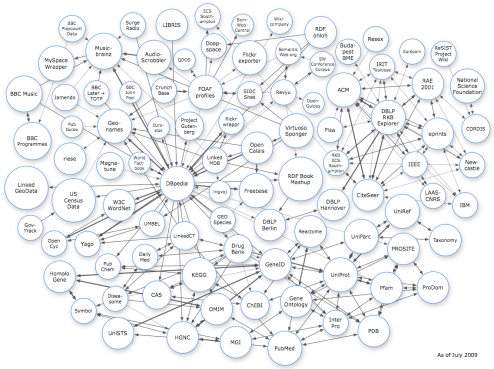
- Many available open data sources in RDF

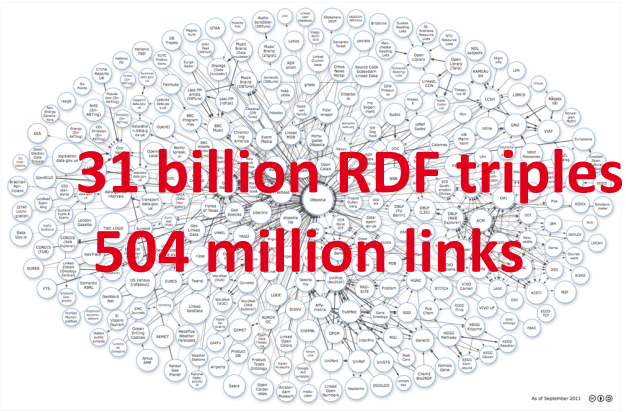
Linked data is growing

LOD cloud









2007 2008 2009 2011

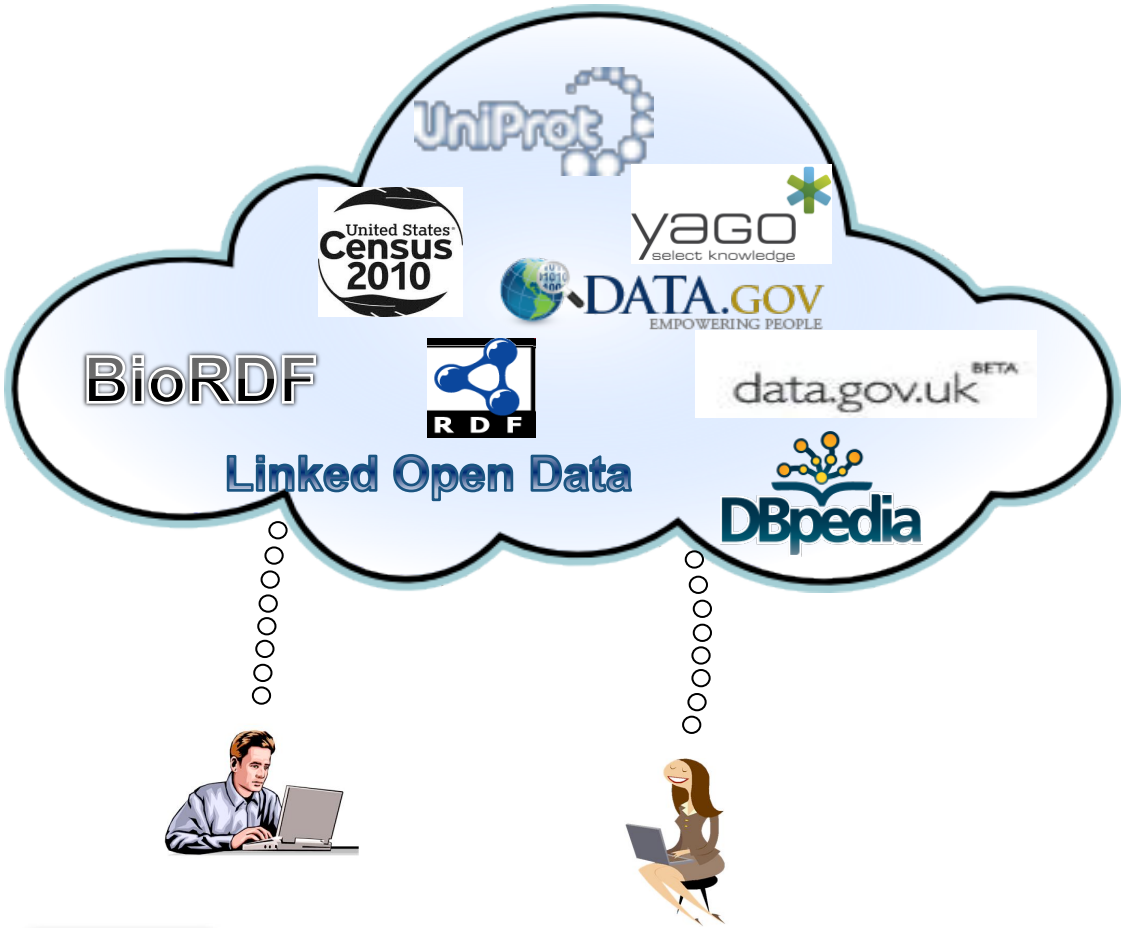
RDF data sources in numbers

LOD cloud	**Linked Open Data cloud**	**31 billion** triples, Sept.
DBpedia	RDF-encoded Wikipedia	1.89 billion triples
BioRDF	RDF-encoded biological data	2.7 billion triples
DATA.GOV EMPOWERING PEOPLE	US government data in RDF	5 billion triples
A new application award Semantic Web Challenge http://challenge.semanticweb.org	crawled Web data	2 billion triples
United States Census 2010	US population statistics	1 billion triples
yago select knowledge	facts from Wikipedia, Wordnet, Geonames	0.12 billion triples

… and many more … http://www.w3.org/wiki/DataSetRDFDumps
http://datahub.io/

Managing large volumes of RDF data

Cloud computing



- Scalability
- Elasticity
- Fault-tolerance

What this seminar is about

Linked data management on the cloud

- Storage RDF data in the cloud
- Evaluation of RDF queries in the cloud
- Inference of RDF data in the cloud

Outline

1. RDF basics
2. Cloud basics
3. Analysis of RDF systems in the cloud
   - Storage
   - Querying
   - Inference
4. Open issues

1

RDF BASICS

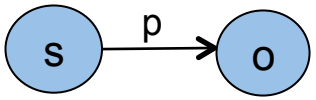
Resource Description Framework (RDF)

- RDF: representation of **resources** on the Web identified by Universal Resource Identifiers (**URIs**) http://live.dbpedia.org/page/Pablo_Picasso **U**
- **Literals**: strings, numbers, dates, etc. **L**
- **Blank nodes**: existential identifiers **B**

- RDF data: facts = **triples** `(s, p, o)`

subject **predicate/ property** **object**

- A well-formed triple is a tuple (s, p, o) from UB × UB × UBL
- RDF as graphs



RDF Schema

- RDF **schema** (RDFS): specifies concepts, properties and relationships between them
  - Concept/class: http://live.dbpedia.org/ontology/Artist
  - Property: http://dbpedia.org/property/birthPlace
  - Relationships:

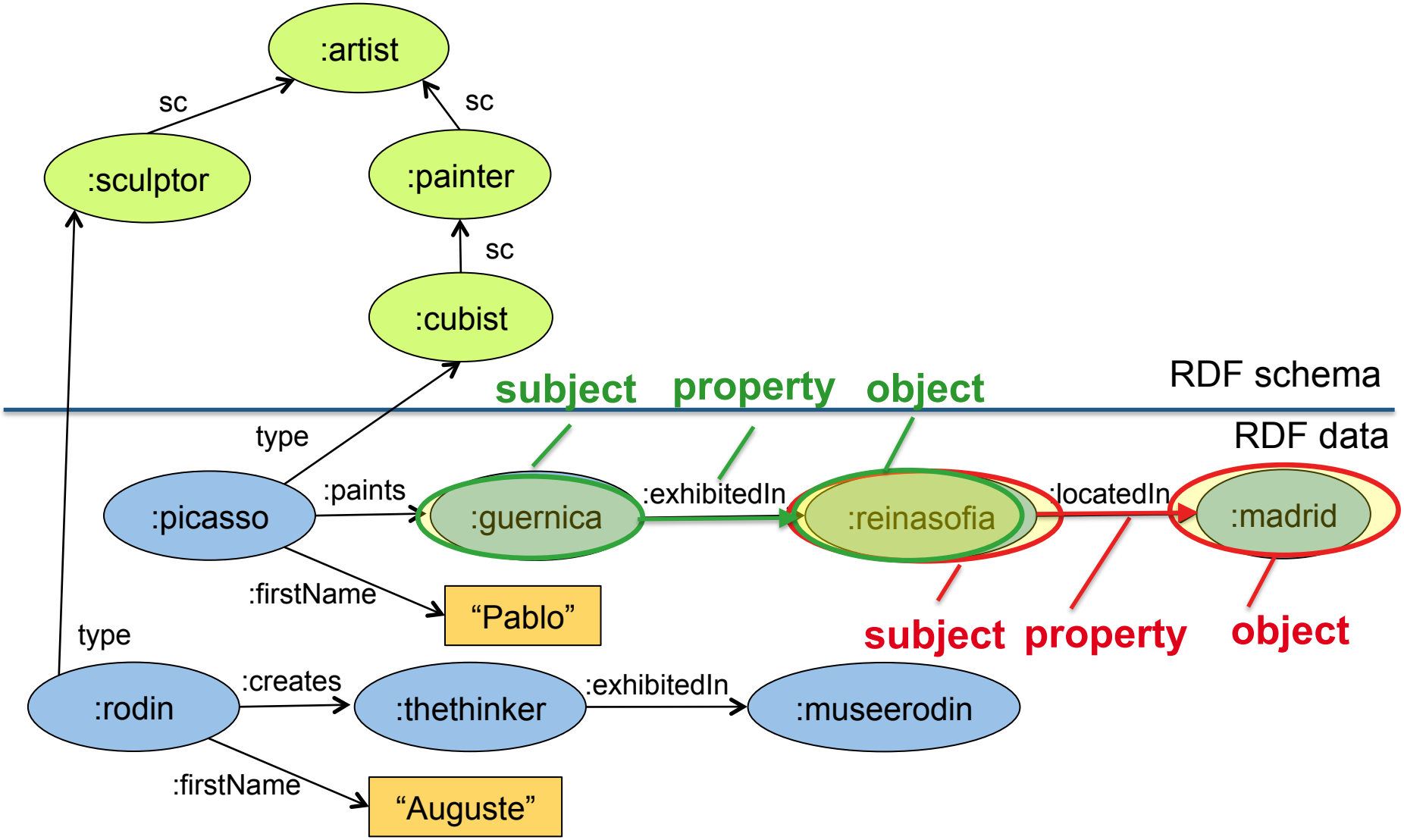
```
(dbpedia-owl:Artist, rdfs:subClassOf, dbpedia-owl:Person)
(dbpprop:birthplace, rdfs:domain, dbpedia-owl:Person)

(dbpedia:Pablo_Picasso, rdf:type, dbpedia-owl:Artist)
```

We skip namespaces and use the following notation for rdf-specific terms:

```
(:Artist, sc, :Person)
(:birtplace, domain, :Person
(:Pablo_Picasso, type, :Artist)
```

Running example: RDF database



Running example: RDF database (triple syntax)

N-TRIPLE syntax

```
:picasso type :cubist .
:picasso :firstName “Pablo” .
:picasso :paints :guernica .
:guernica :exhibitedIn :reinasofia .
:reinasofia :locatedIn :madrid .
:rodin type :sculptor .
:rodin :firstName “Auguste” .
:rodin :creates :thethinker .
:thethinker :exhibitedIn :museerodin .
```

RDF data

```
:sculptor sc :artist .
:painter sc :artist .
:cubist sc :painter.
```

RDF Schema

SPARQL Query Language

- Basic building block
  - Triple pattern (s, p, o) from UBV × UBV × UBLV

constant or variable

U: URIs
B: Bnodes
L: literals
V: variables

- SQL-based syntax
  - Basic graph pattern (BGP) queries:

SELECT $?x_1$, $?x_2$, …, $?x_n$ → **distinguished variables**

WHERE {

s_1 p_1 o_1 .

…

s_k p_k o_k .

}

Conjunction of triple patterns

SPARQL Query Language

- BGP queries in conjunctive form:
  - $?x_1, ?x_2, \ldots, ?x_n : (s_1, p_1, o_1) \wedge \ldots \wedge (s_k, p_k, o_k)$

distinguished variables — **triple patterns**

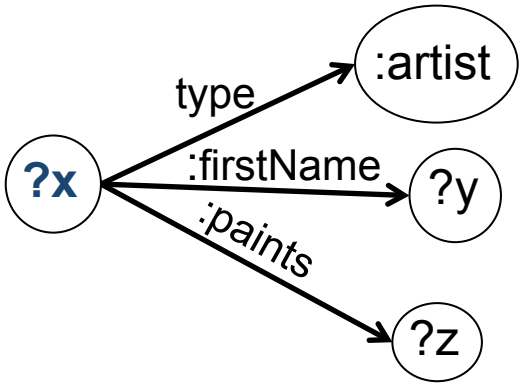
- BGP queries: Select-Project-Join fragment of relational algebra
  - Select → ?x: (?x, :paints, ?y)
  - Project → ?x: (?x, :paints, ?y)
  - Join → ?x: (?x, :paints, ?y) ∧ (?y, :exhibited, :reinasofia)

- More features in SPARQL 1.1: W3C recommendation, March 2013

SPARQL representative queries

Star query

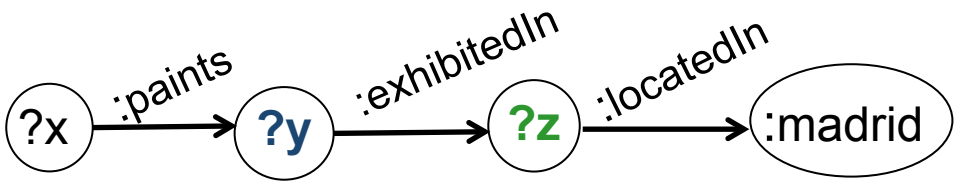
```
SELECT ?y ?z
WHERE {?x type :artist .
       ?x :firstName ?y .
       ?x :paints ?z .}
```



?y ?z: (?x type :artist ∧
(?x :firstName ?y) ∧
(?x :paints ?z)

Path query

```
SELECT ?x ?y ?z
WHERE {?x :paints ?y .
       ?y :exhibitedIn ?z .
       ?z :locatedIn :madrid.}
```



?x ?y ?z: (?x :paints ?y) ∧
(?y :exhibitedIn ?z) ∧
(?z :locatedIn :madrid)

Centralized RDF stores

- How do we usually store RDF in centralized environment?
- Approaches relying on RDBMs
  - Big **triple tables**

Triples

subject	property	object
:picasso	type	:cubist
:picasso	:firstName	“Pablo”
picasso	:paints	:guernica
:guernica	:exhibitedIn	:reinasofia
:reinasofia	:locatedIn	:madrid
:rodin	type	:sculptor
:rodin	:creates	:thethinker
…	…	…

Centralized RDF stores

- How do we usually store RDF in centralized environment?
- Approaches relying on RDBMs
  - Big triple tables
  - **Property tables** [Alexaki01, Wilkison06]

Artists

subject	type	firstName	paints	creates
:picasso	:cubist	“Pablo”	:guernica	null
:rodin	:sculptor	“Auguste”	null	:thethinker

Artifacts

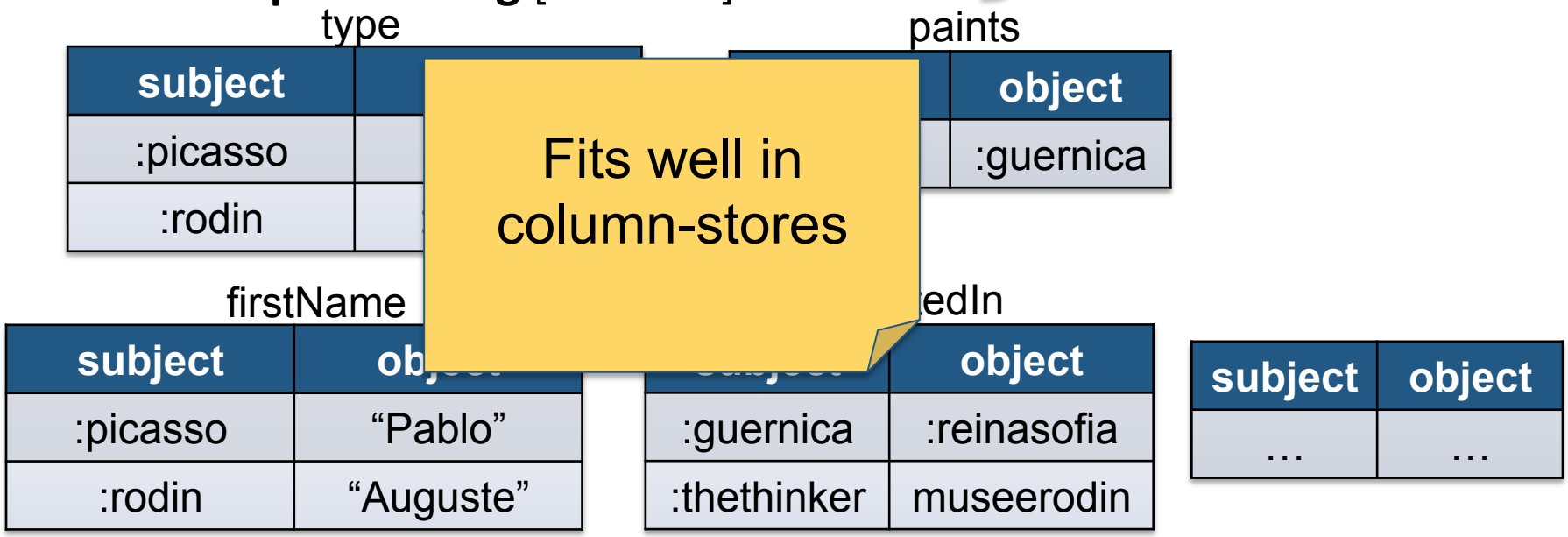
subject	exhibitedIn
:guernica	:reinasofia
:thethinker	:museerodin

subject	…
…	…
…	…

Centralized RDF stores

- How do we usually store RDF in centralized environment?
- Approaches relying on RDBMs
  - Big triple tables
  - Property tables [Alexaki01, Wilkison06]
  - **Vertical partitioning** [Abadi07]

Benchmarking [Theoharis05]



Centralized RDF stores

- How do we usually store RDF in centralized environment?
- Approaches relying on RDBMs
  - Big triple tables
  - Property tables [Alexaki01, Wilkison06]
  - Vertical partitioning [Abadi07]
- Native RDF stores
  - RDF-3X [Neumann10], Hexastore [Weiss08]
  - 6 indexes (1 for each permutation of s p o)
  - compressed indices
  - aggregated indices

Dictionary encoding

- RDF contains long strings of URIs

Size of triples

min	max	avg
80 bytes	2100 bytes	240 bytes

Source http://gromgull.net/blog/category/semantic-web/billion-triple-challenge/

Dictionary

id	resource
1	:picasso
2	:rodin
3	type
4	:cubist
5	:firstName
6	:guernica
…	…

Triples

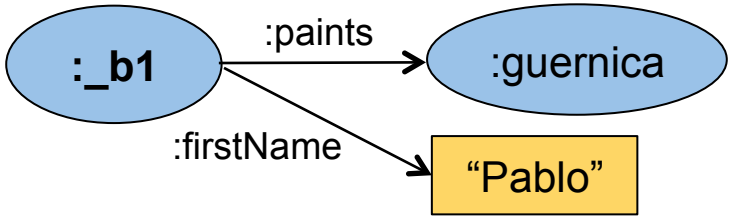
pro	
t	
:firs	
:p	
:exh	
:loc	
type	:sculptor
:creates	:thethinker
…	…

Triples

subject	property	object
1	3	4
1	5	10
1	7	6
6	12	9
…	…	…

RDF is not just a 3-attribute table

- Blank nodes
  - unknown/anonymous resources
  - existential variables



  - for more “On blank nodes” [Mallea11]
- Schema queries
  - ?x: (?x, sc, :artist) ∧ (:paints, domain, ?x)
- Data and schema can be queried together
  - ?x: (?x, :paints, ?y) ∧ (?x, type, ?z) ∧ (?z, sc, :artist)

Data query

Schema query

RDF is not just a 3-attribute table

- Properties in RDF are treated as first class citizens (they are also resources)
  - `(:paints, domain, :painter)`
  - joins between properties and subjects/objects

    `?x: (?x, ?y, :guernica)` $\wedge$ `(?y, domain, :painter)`
- RDFS inference
- and more …

Inference in RDF

- RDF schema statements and set of entailment rules lead to implicit (entailed) RDF data

Sample rules

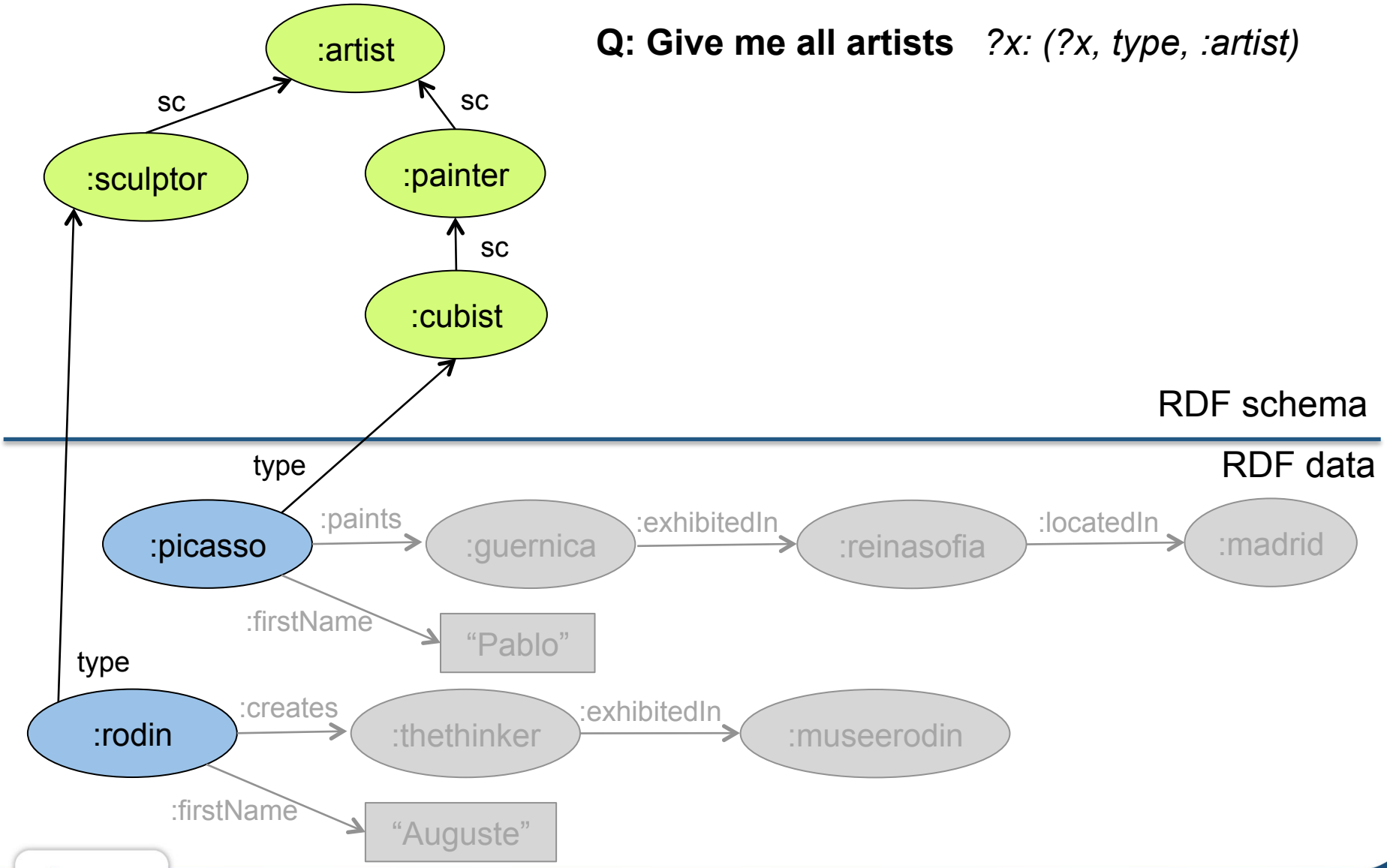
$(X, type, A) \wedge (A, sc, B) \rightarrow (X, type, B)$
$(X, P, O) \wedge (P, domain, A) \rightarrow (X, type, A)$

(:picasso, :paints, :guernica) ∧ (:paints, domain, :painter) → (:picasso, type, :painter)

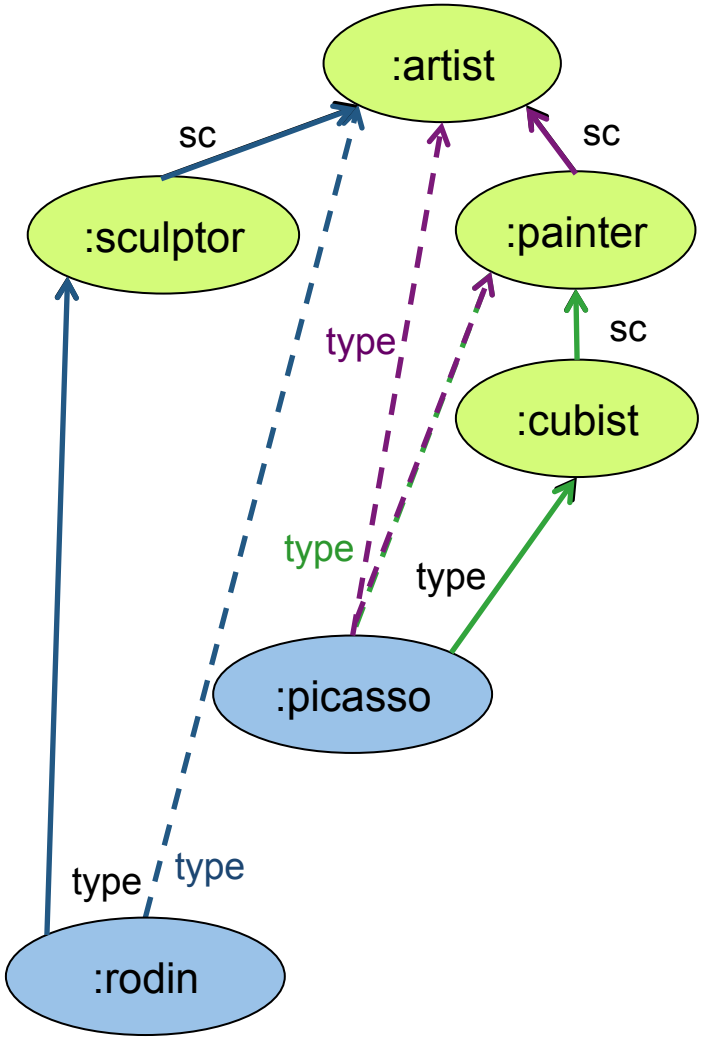
- RDFS entailment : *Given an RDF(S) database DB and a triple t, is t entailed from DB (or does t logically follows from DB)?*

Entailed triples and RDF querying

Q: Give me all artists *?x: (?x, type, :artist)*



Entailed triples and RDF querying



Q: Give me all artists *?x: (?x, type, :artist)*

A:

?x
-

no entailment

Rule:

(X, type, A) ∧ (A, sc, B) → (X, type, B)

Implicit / entailed triples

(:rodin, type, :artist)

(:picasso, type, :painter)

(:picasso, type, :artist)

A:

?x
:picasso
:rodin

with entailment

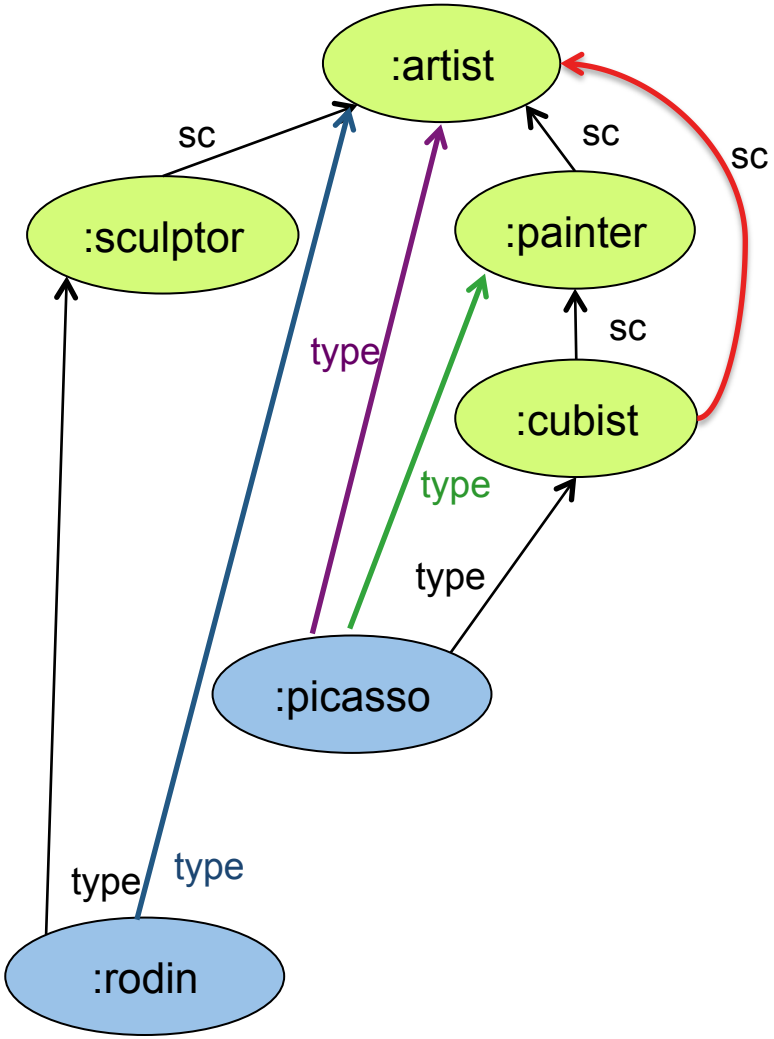
RDFS entailment techniques

- RDFS closure
  - compute all implicit triples and store/materialize them in the database
- Query reformulation
  - transform/rewrite the query so that you get all the answers (even the ones that come from implicit triples)
- Hybrid approaches
  - compute the schema closure, and then reformulate the query
  - magic sets: based on the queries, precompute and store only the implicit triples that are required for the answer

RDFS entailment – RDFS closure

Before querying

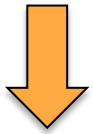
subject	property	object
:picasso	type	:cubist
:rodin	type	:sculptor
:sculptor	sc	:artist
:cubist	sc	:painter
:painter	sc	:artist
:cubist	**sc**	**:painter**
:rodin	**type**	**:artist**
:picasso	**type**	**:painter**
:picasso	**type**	**:artist**



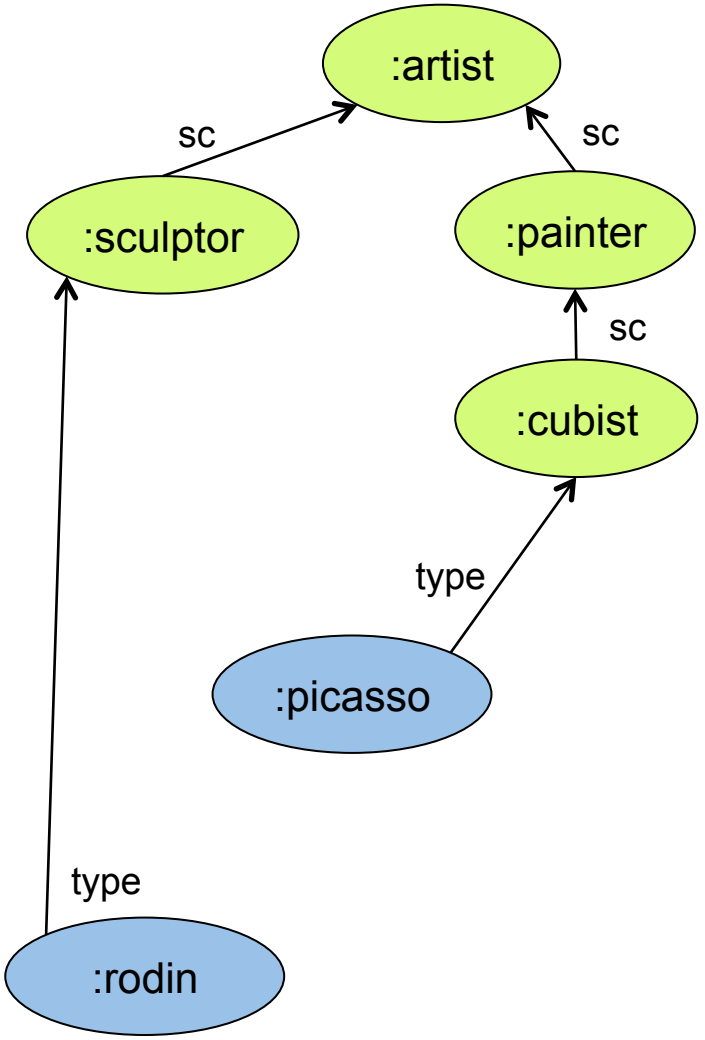
RDFS entailment – query reformulation

At query time

?x: (?x, type, :artist)



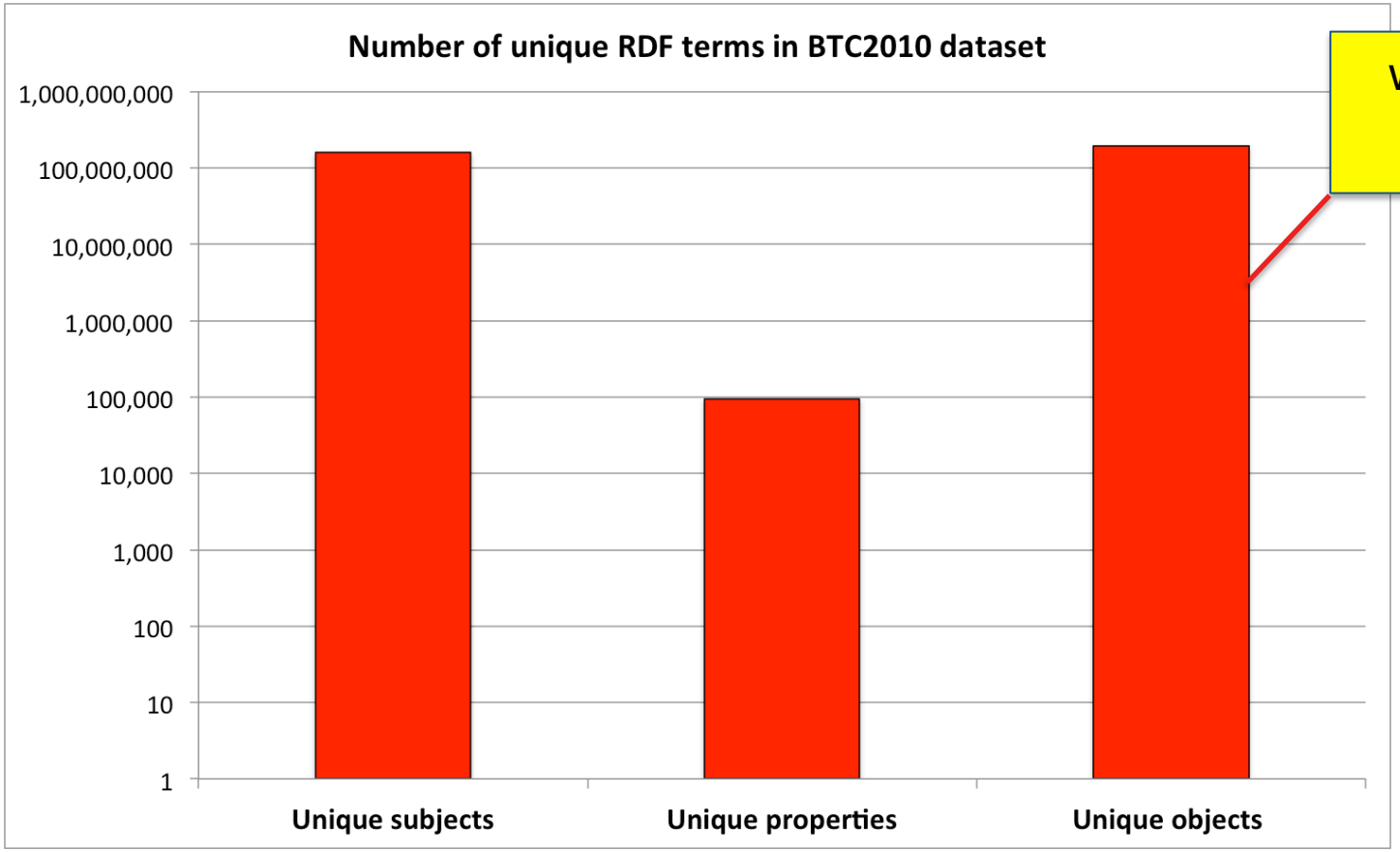
?x: (?x, type, :artist) ⋁
(?x, type, :sculptor) ⋁
(?y, type, :painter) ⋁
(?x, type, :cubist)



RDFS entailment techniques

- RDFS closure
  - ✚ Simple query evaluation
  - ▬ A lot of storage overhead
  - ▬ Data maintenance upon updates
- Query reformulation
  - ✚ No storage overhead
  - ✚ Robust to data/schema changes
  - ▬ Not efficient query evaluation for complex queries
- Hybrid approaches
  - in the middle of the two
- For a comparison on these techniques:
  - centralized [Goasdoué13] and distributed [Kaoudi13]

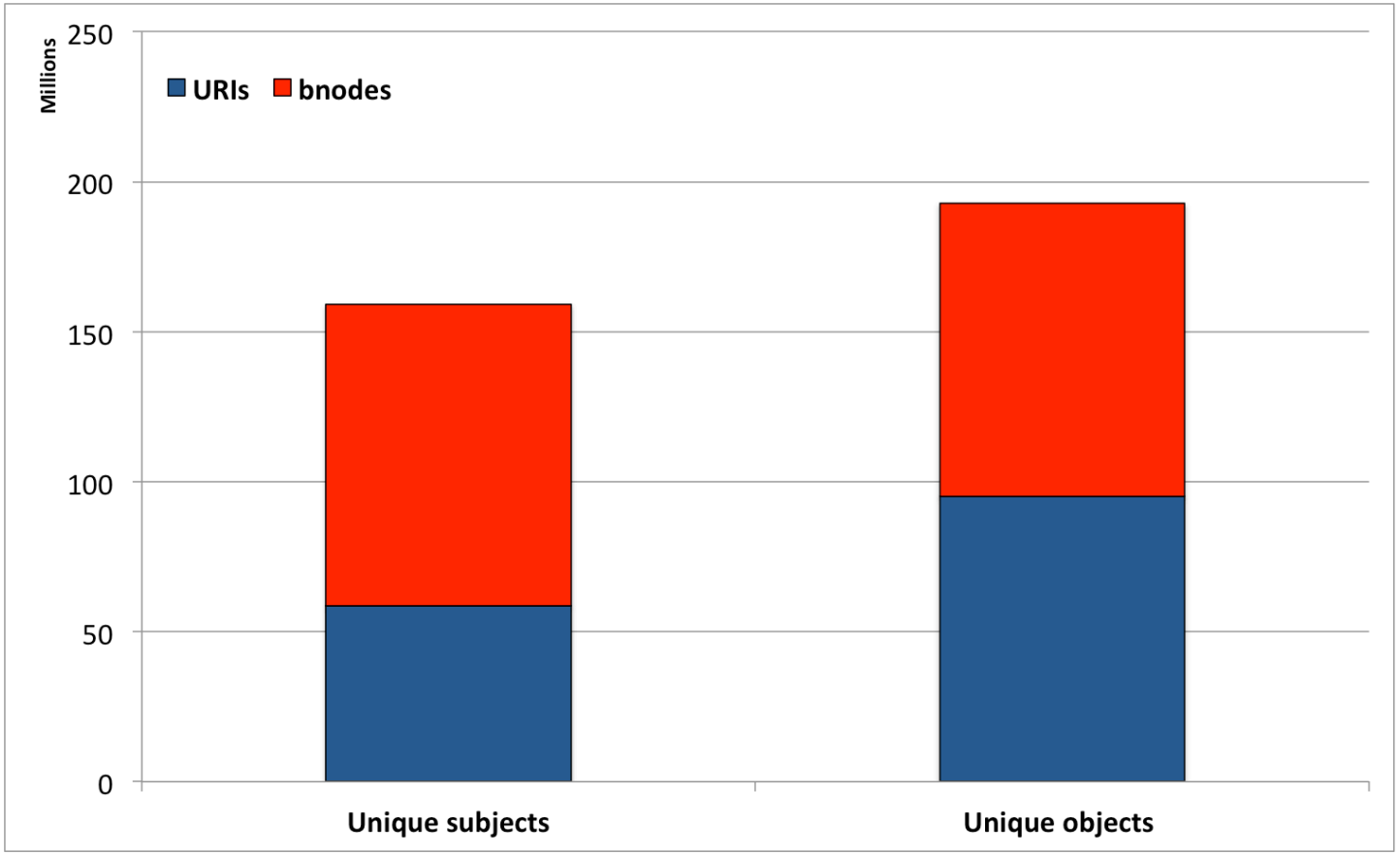
How does real RDF look?



Data taken from http://gromgull.net/blog/2010/09/btc2010-basic-stats/

How does real RDF look?

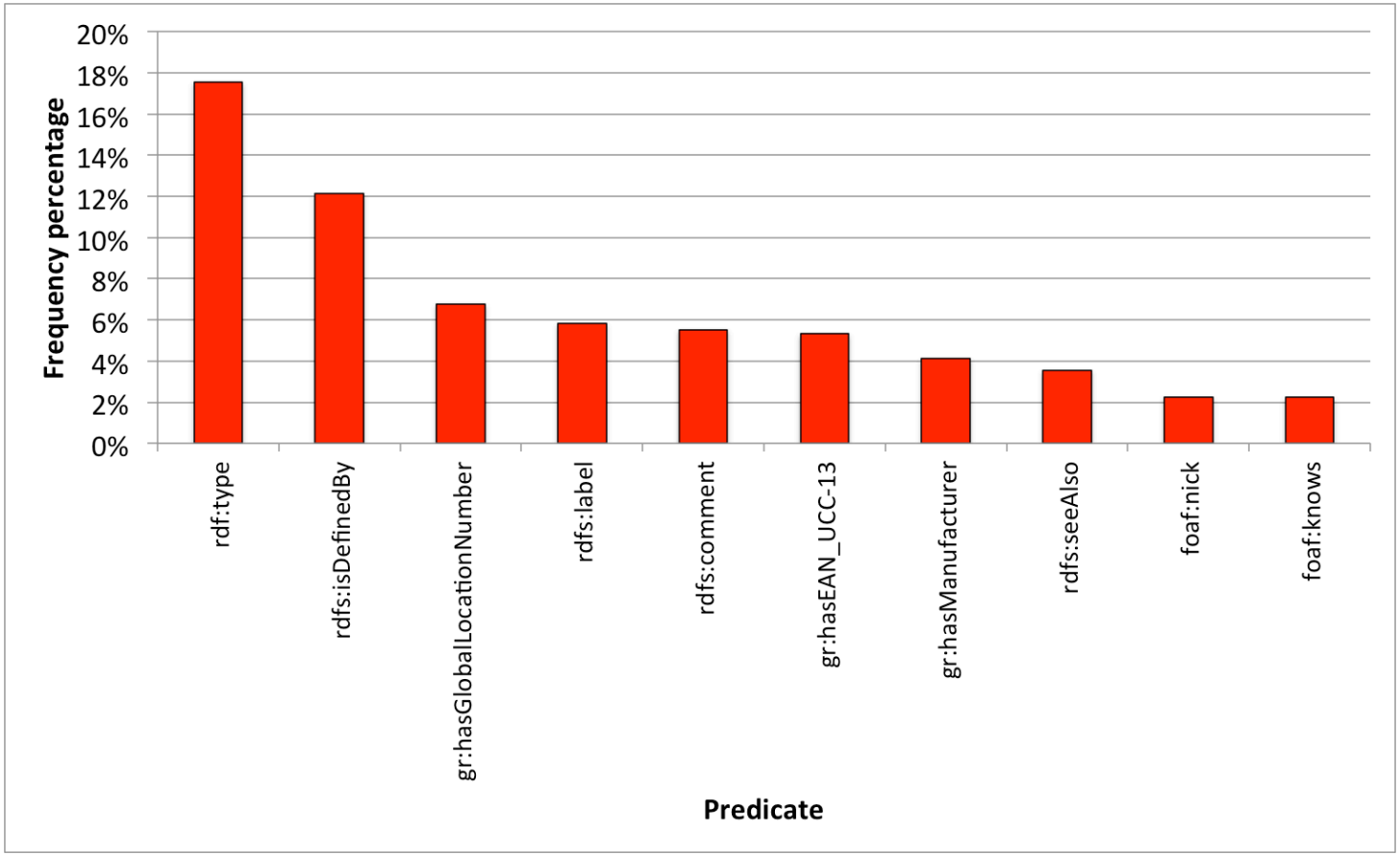
BTC2010 dataset



Data taken from http://gromgull.net/blog/2010/09/btc2010-basic-stats/

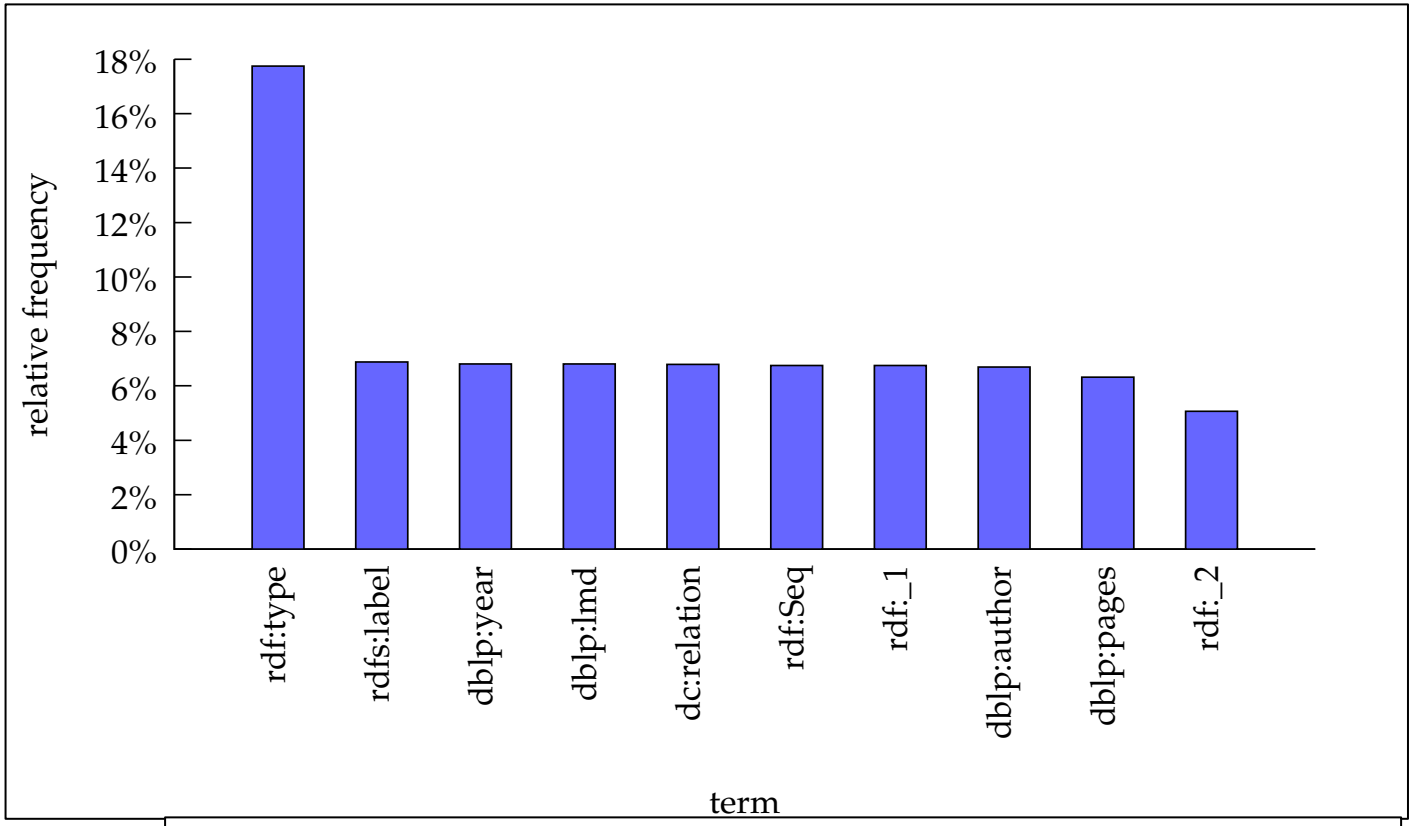
How does real RDF look?

BTC2010 dataset



Data taken from http://gromgull.net/blog/2010/09/btc2010-basic-stats/

How does real RDF look?



Top-10 terms of SwetoDBLP [Kotoulas10]

2

CLOUD BASICS

Cloud computing

The origin of the term cloud computing was derived from [...] drawings of stylized clouds to denote networks [...]
The cloud symbol was used to represent the Internet as early as 1994.
[Wikipedia: http://en.wikipedia.org/wiki/Cloud_computing]

- Use of computing resources as a service over the network
- Scalability
- Elasticity
- Reliability

Cloud computing service models

Google Apps

Software-as-a-Service (SaaS)

Applications

Windows Azure

web sites

Platform-as-a-Service (PaaS)

Platforms

amazon web services™

Elastic Compute Cloud (EC2)

Infrastructure-as-a-Service (IaaS)

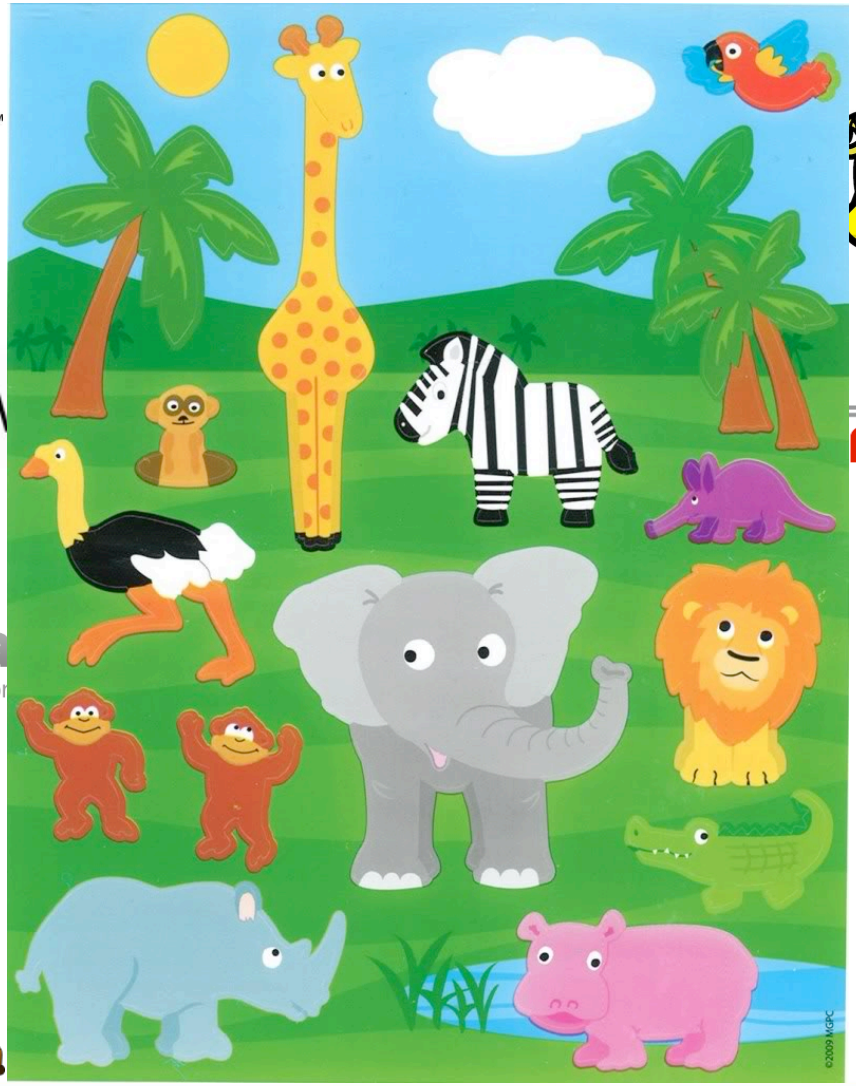
Hardware

pay-per-use

Cloud computing and data management

- Relational databases in the cloud (DBaaS)
  - IBM DB2 on SmartCloud
  - Amazon RDS (MySQL, Oracle or SQL Server)
- NoSQL systems [Catell11]:
  - Key-value stores (memcached, DynamoDB, SimpleDB)
  - Extensible record stores (BigTable, Cassandra, HBase, Accumulo)
  - Document stores (MongoDB, CouchDB)
  - Graph databases (Pregel, Apache Giraph, Neo4J)
- Parallel data processing paradigms:
  - MapReduce (Hadoop, Amazon EMR), PACTs/Stratosphere

Cloud ecosystem



Cloud infrastructures

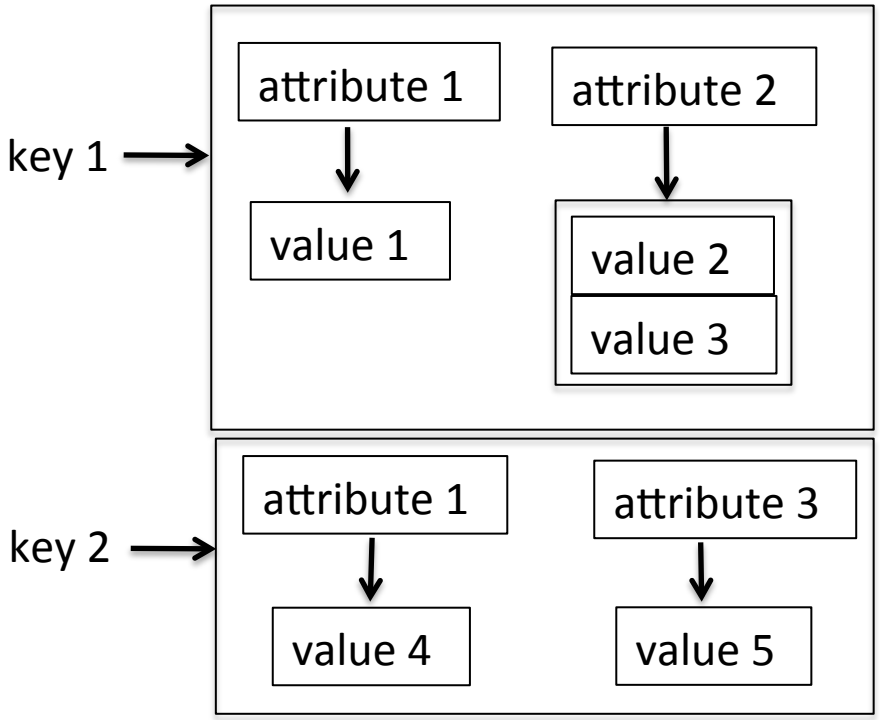
I. Distributed key-value stores

II. Parallel processing with MapReduce

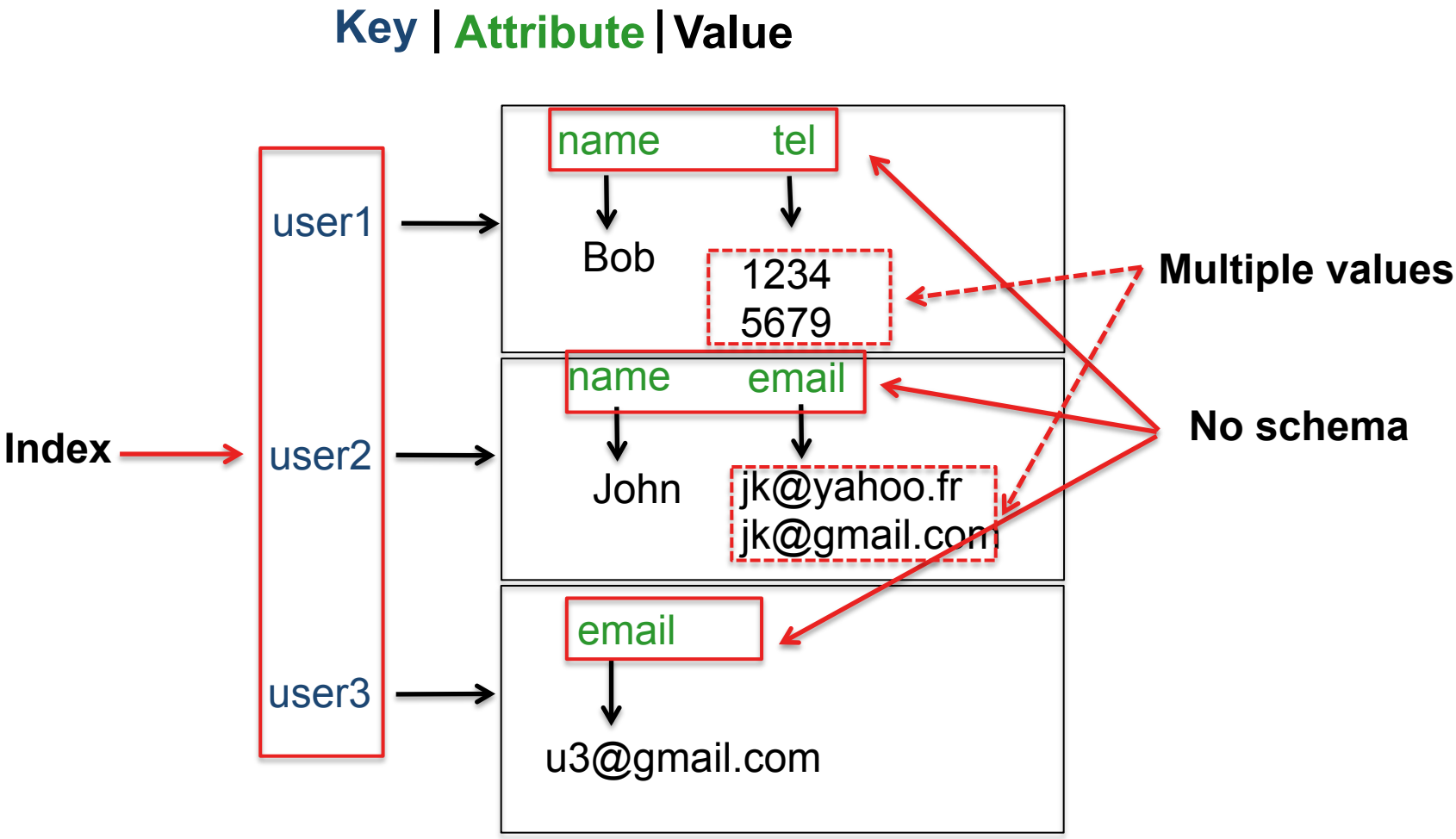
Distributed key-value stores

- Storing schema-less data in key-value pairs
- For each key there may be multiple (attribute-value) pairs
- Multi-value attributes
- Index: Hash-based or lexicographically sorted on the key
- Basic APIs:

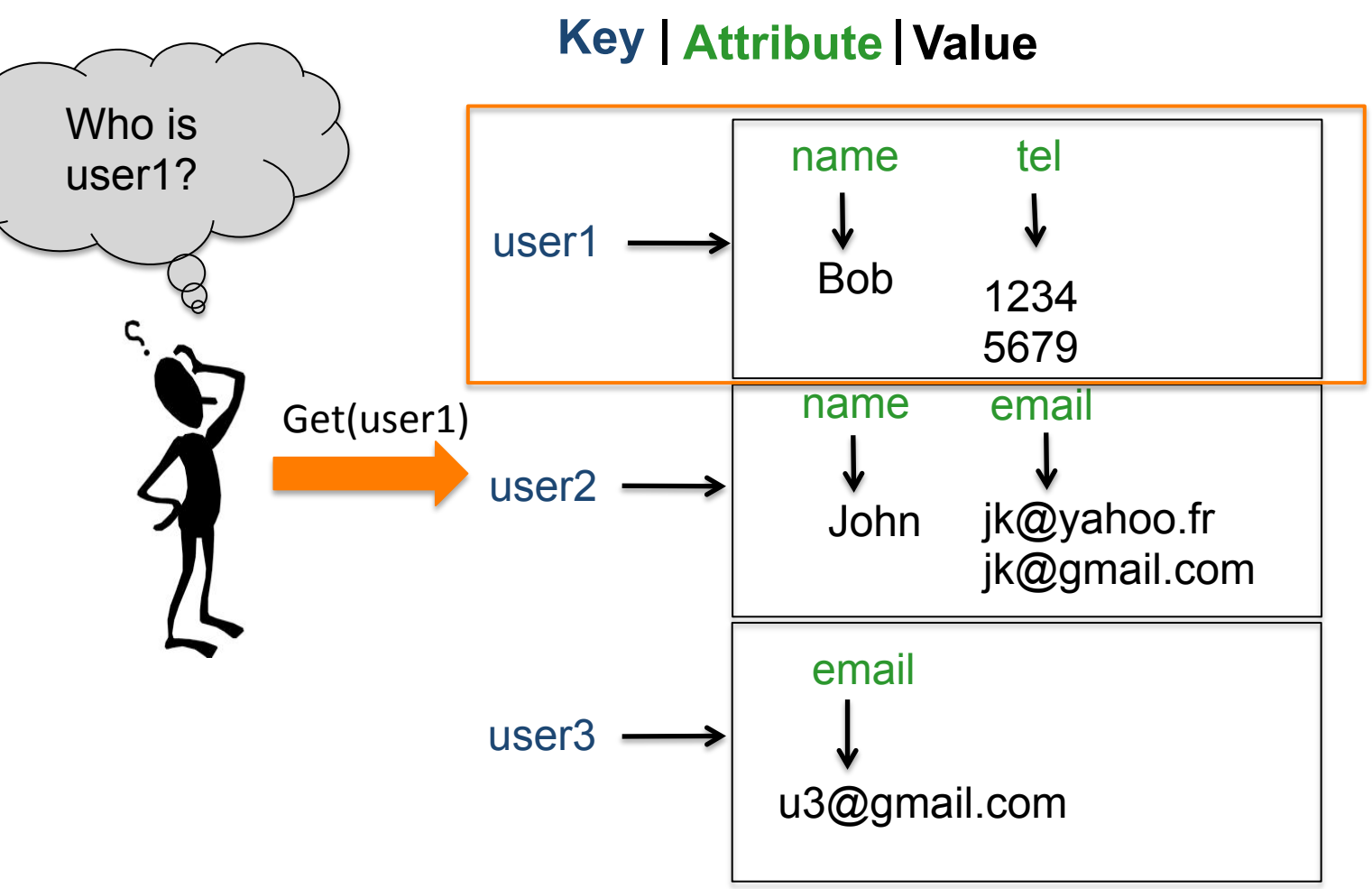
  Put(k, <a, v>), Get(k), Delete(k)
- No support for joins or any operations across different tables



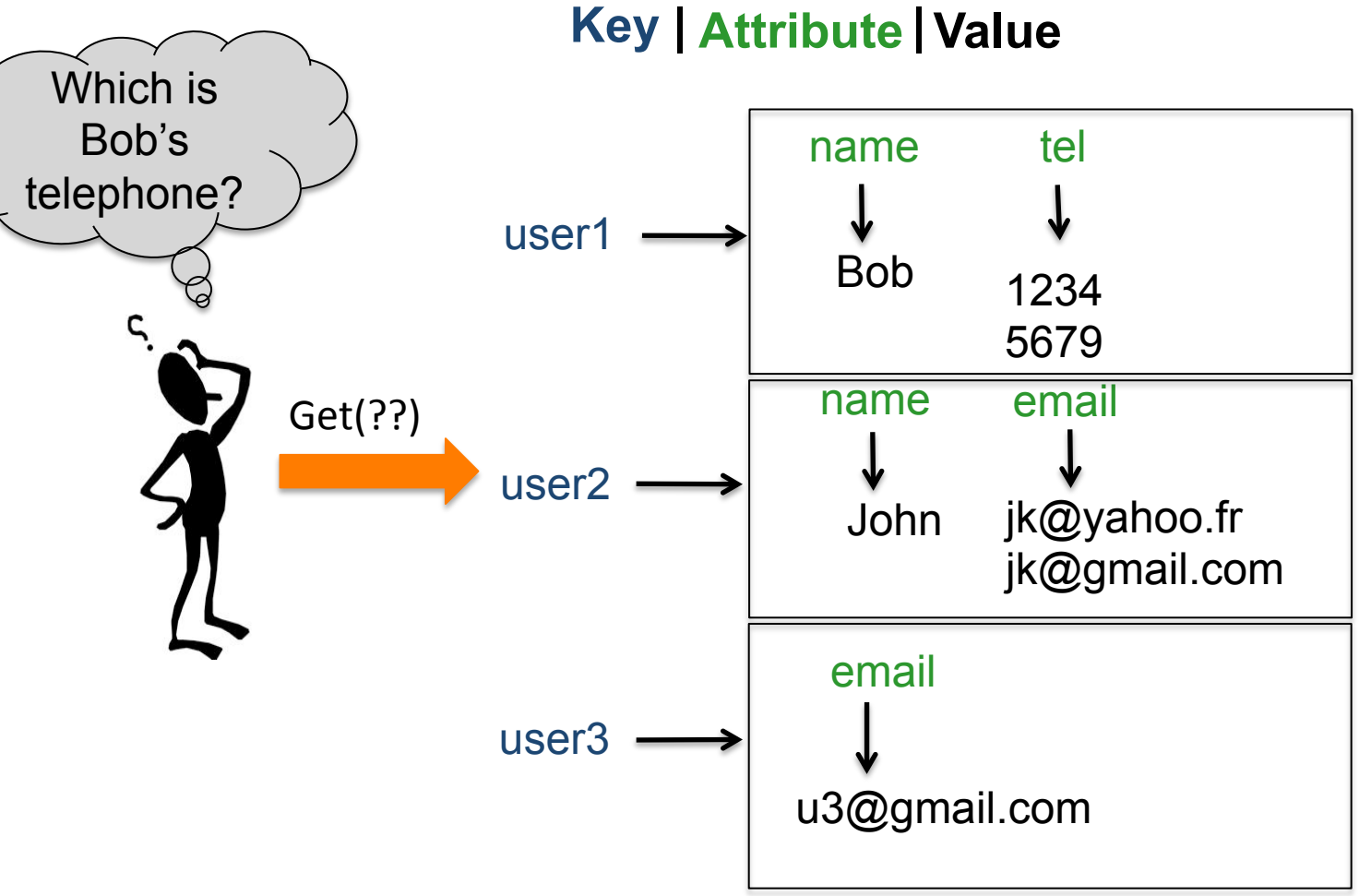
Distributed key-value stores



Distributed key-value stores

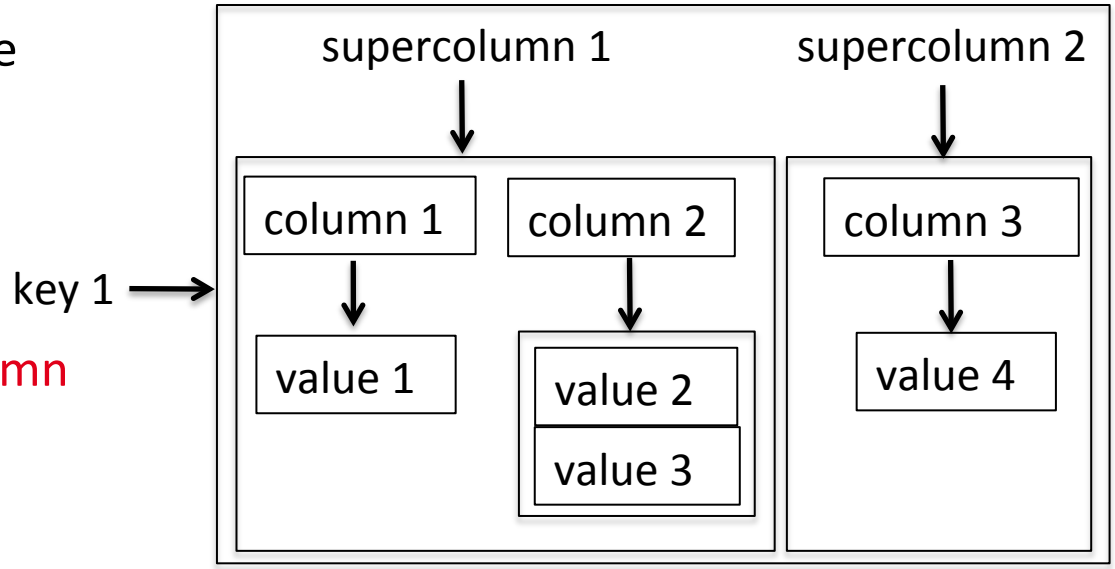


Distributed key-value stores



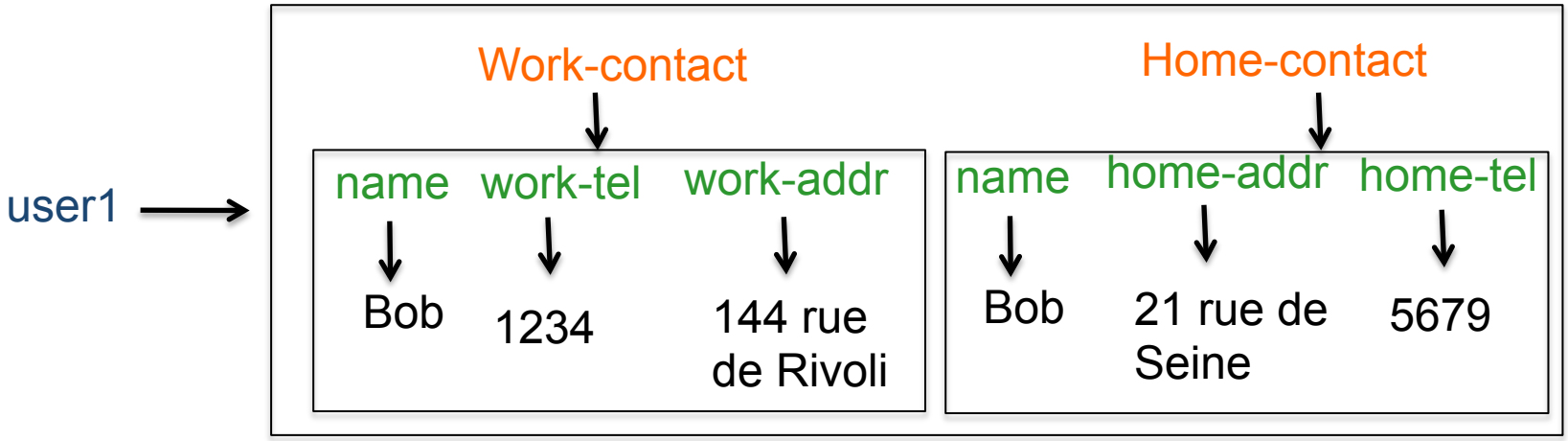
Extended key-value stores

- Google BigTable, Apache Cassandra
- Supercolumns: columns whose values are columns
  - K|{SC}C|V
- Sorted index on the key
- Sorted index on the supercolumn
- Secondary index
  - maps values to keys



Extended key-value stores

Key | {SuperColumn}Column | Value



Cloud infrastructures

I. Distributed key-value stores

II. Parallel processing with MapReduce

MapReduce in a nutshell

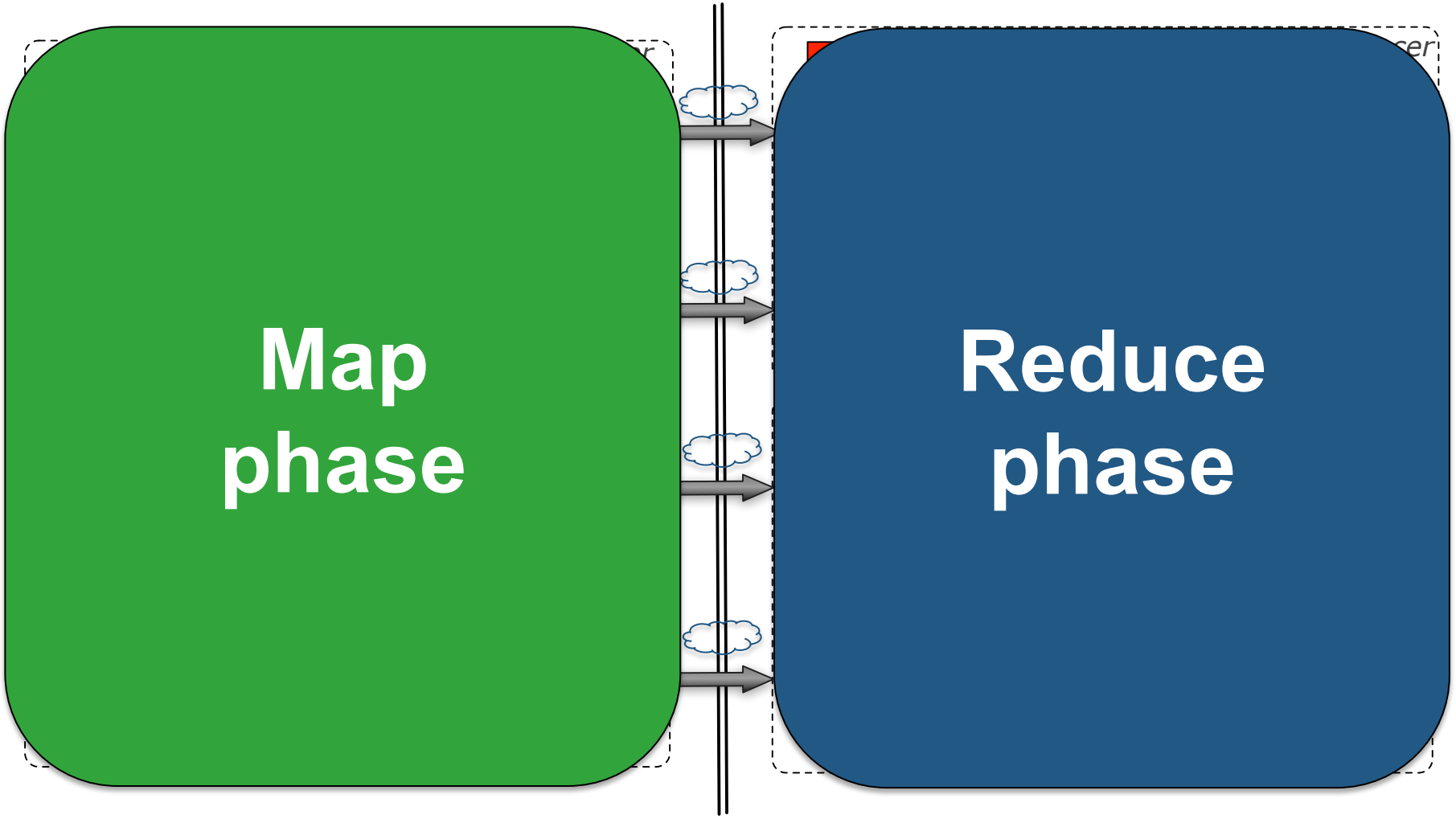
- Google's proposal [DG04]
- Massively parallel processing on commodity machines
- Master/slave architecture
- Input data is split horizontally into chunks
  - distributed file system (DFS)
- Simple API:
  - Map (*key*, *value*) → {*ikey*, *ivalue*}
  - Reduce (*ikey*, {*ivalues*}) → (*key'*, *value'*)

MapReduce in a nutshell

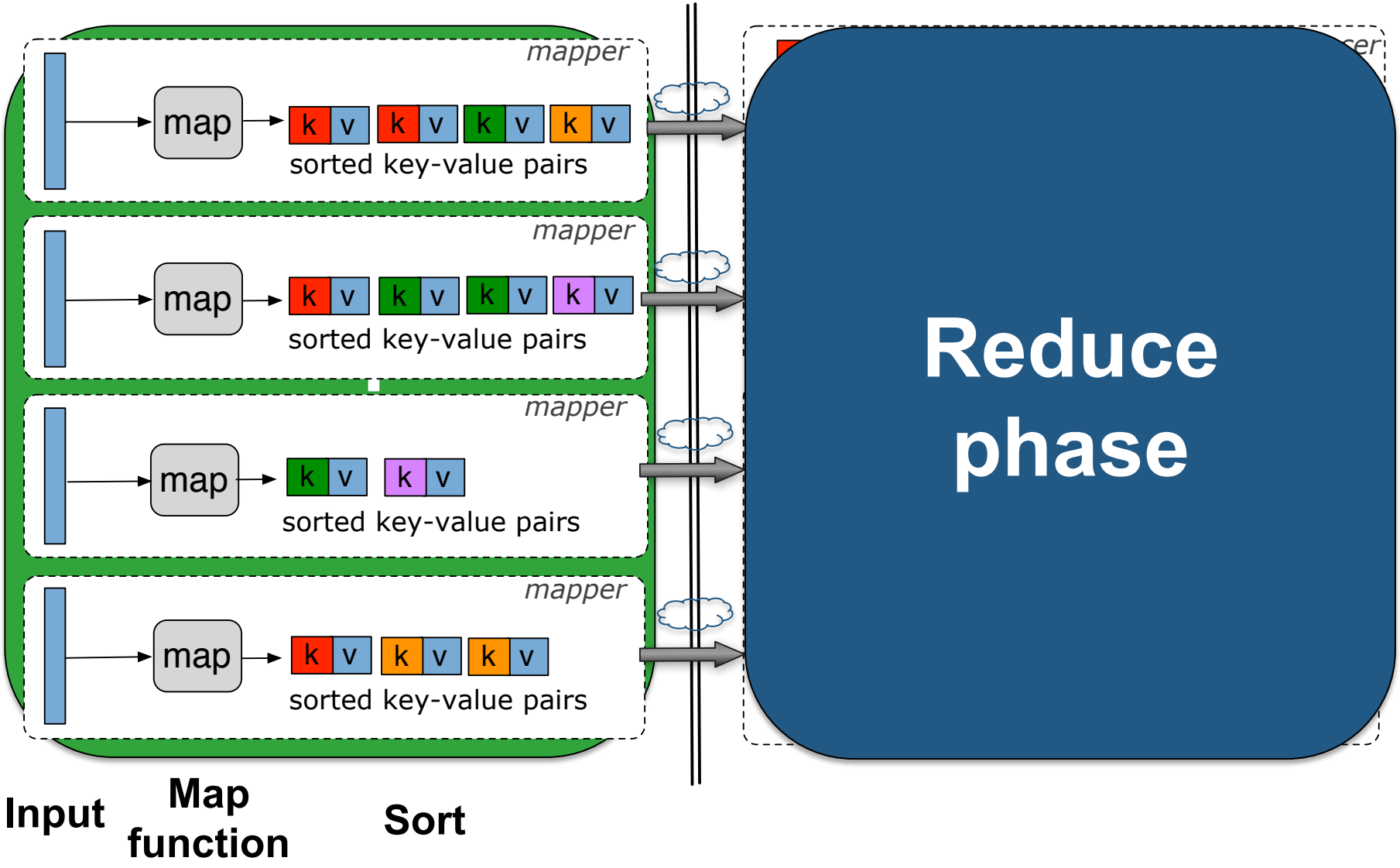
- **Map** phase: independent processes (mappers) which run in parallel
  - operate on the chunks of input data
  - output intermediate results
- **Shuffle** phase: Intermediate results are shuffled through the network
- **Reduce** phase: independent processes (reducers) which run in parallel
  - group intermediate results of the map phase
  - operate on the groups
  - output final results

- Key-value pairs with the same key *ikey* meet at the same reducer
  - e.g., hash(*ikey*) mod R
- Open source implementation: Hadoop and HDFS

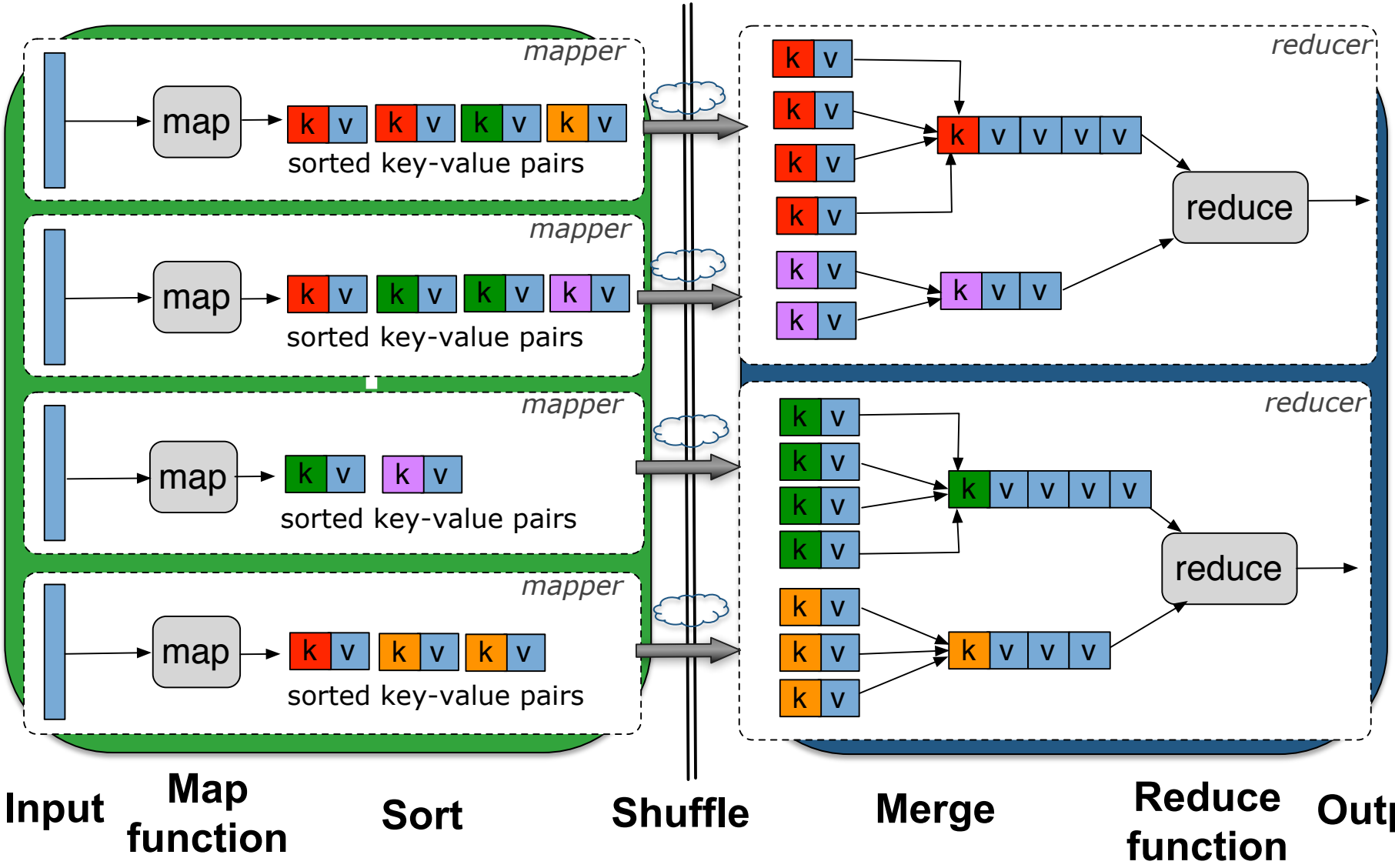
MapReduce illustration



MapReduce illustration

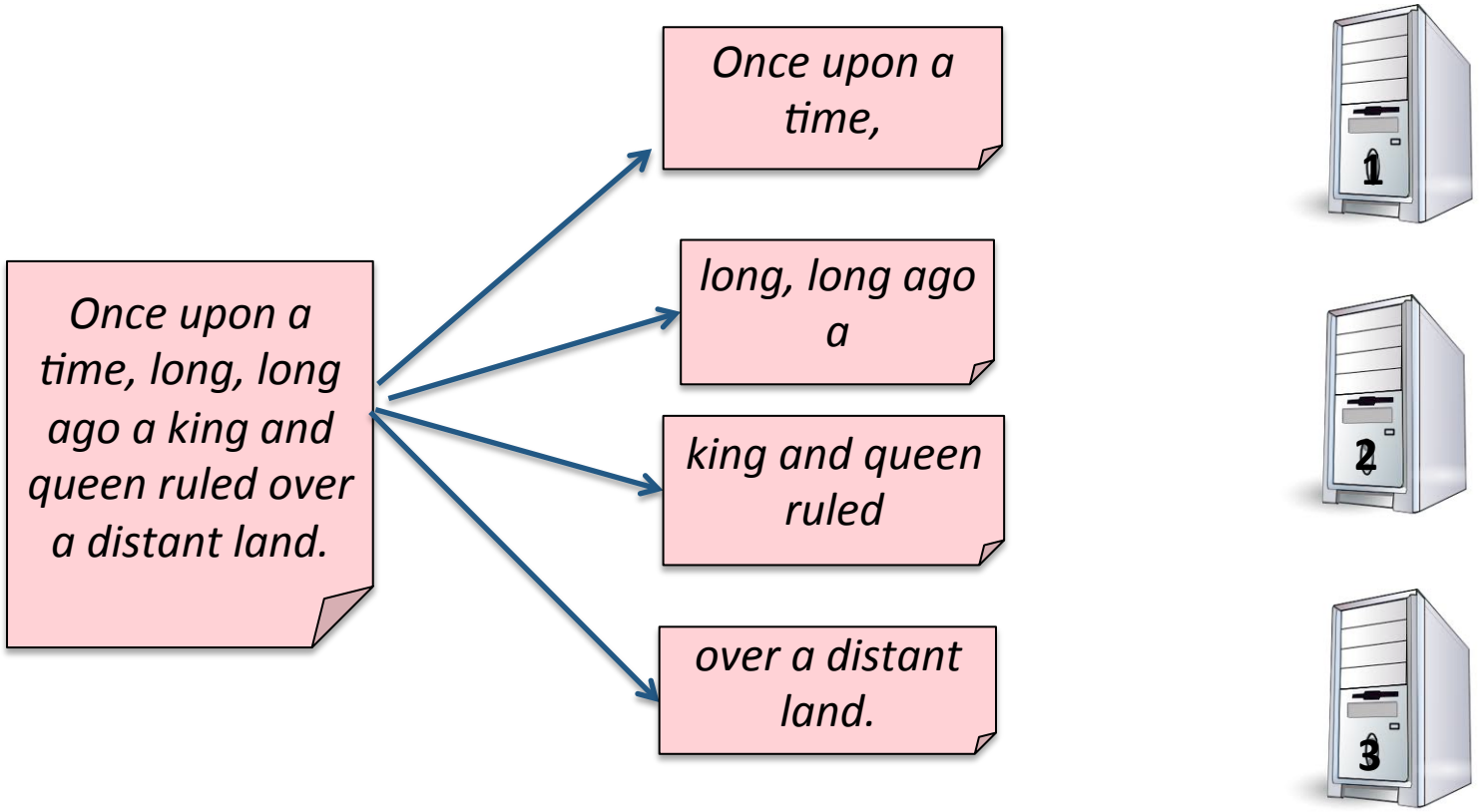


MapReduce illustration



MapReduce by words-count example

- *Count the number of words in parallel*

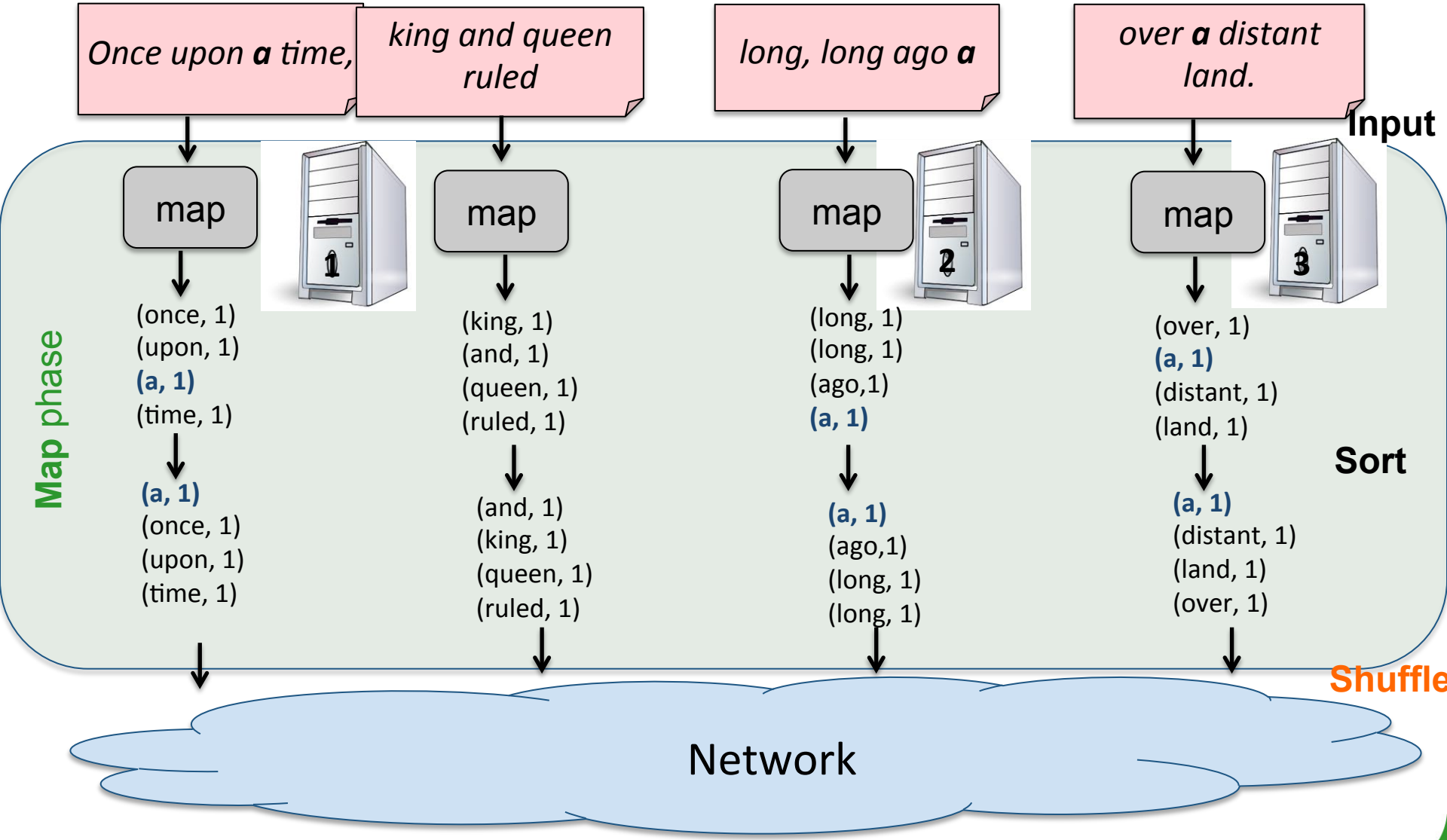


Initial file

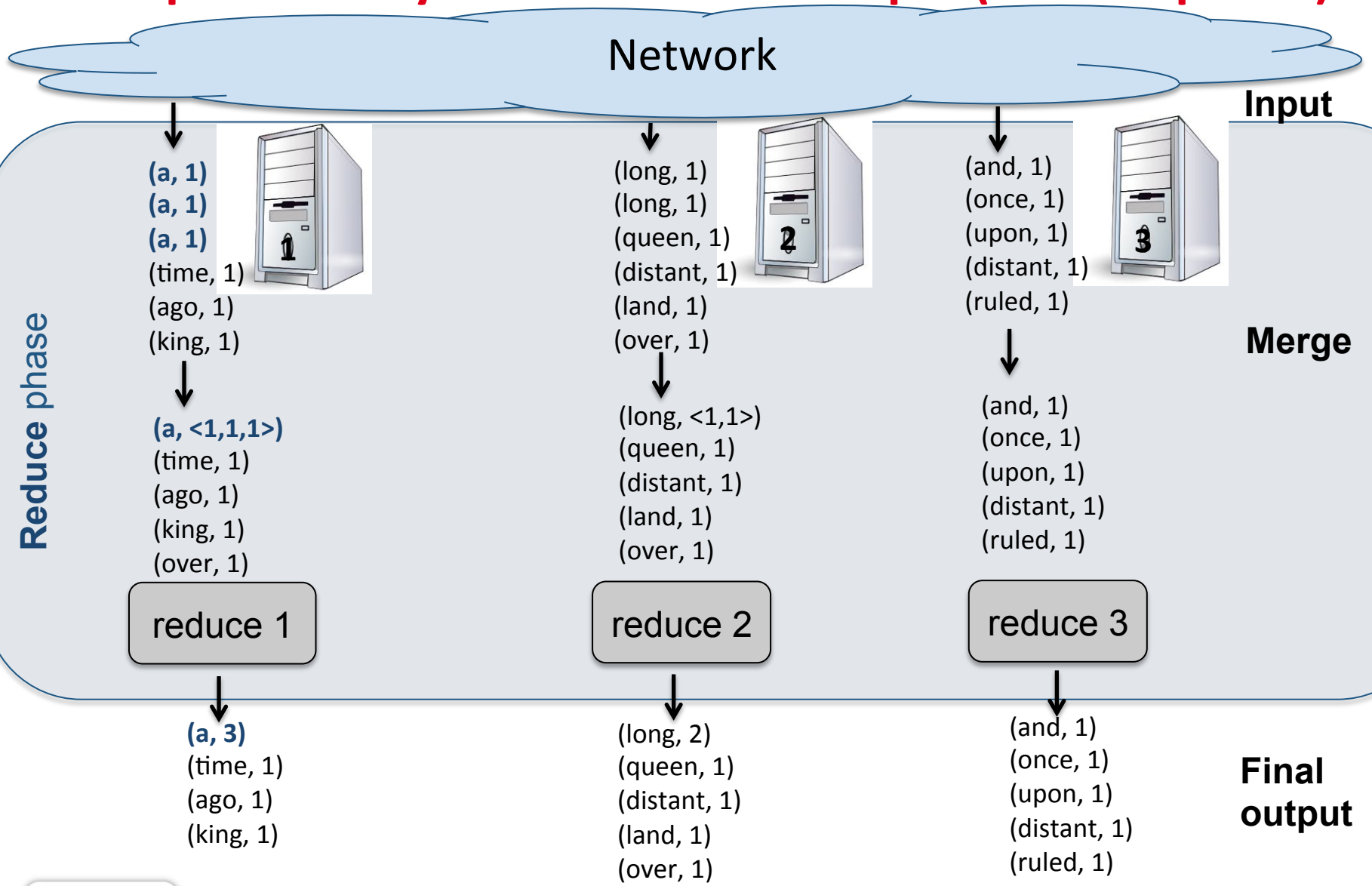
split by DFS in *data blocks*

3-nodes cluster

MapReduce by words-count example (Map phase)



MapReduce by words-count example (Reduce phase)



MapReduce in code ☺

```
1 package org.myorg;
2
3 import java.io.IOException;
4 import java.util.*;
5
6 import org.apache.hadoop.fs.Path;
7 import org.apache.hadoop.conf.*;
8 import org.apache.hadoop.io.*;
9 import org.apache.hadoop.mapreduce.*;
10 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
11 import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
12 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
13 import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
14
15 public class WordCount {
16
17  public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
18    private final static IntWritable one = new IntWritable(1);
19    private Text word = new Text();
20
21    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
22      String line = value.toString();
23      StringTokenizer tokenizer = new StringTokenizer(line);
24      while (tokenizer.hasMoreTokens()) {
25        word.set(tokenizer.nextToken());
26        context.write(word, one);
27      }
28    }
29  }
30
31  public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
32
33    public void reduce(Text key, Iterable<IntWritable> values, Context context)
34      throws IOException, InterruptedException {
35      int sum = 0;
36      for (IntWritable val : values) {
37        sum += val.get();
38      }
39      context.write(key, new IntWritable(sum));
40    }
41  }
42
43  public static void main(String[] args) throws Exception {
44    Configuration conf = new Configuration();
45
46      Job job = new Job(conf, "wordcount");
47
48    job.setOutputKeyClass(Text.class);
49    job.setOutputValueClass(IntWritable.class);
50
51    job.setMapperClass(Map.class);
52    job.setReducerClass(Reduce.class);
53
54    job.setInputFormatClass(TextInputFormat.class);
55    job.setOutputFormatClass(TextOutputFormat.class);
56
57    FileInputFormat.addInputPath(job, new Path(args[0]));
58    FileOutputFormat.setOutputPath(job, new Path(args[1]));
59
60    job.waitForCompletion(true);
61  }
62
63 }
```

http://wiki.apache.org/hadoop/WordCount

Only **63** lines for a distributed/parallel application!

MapReduce - pros/cons

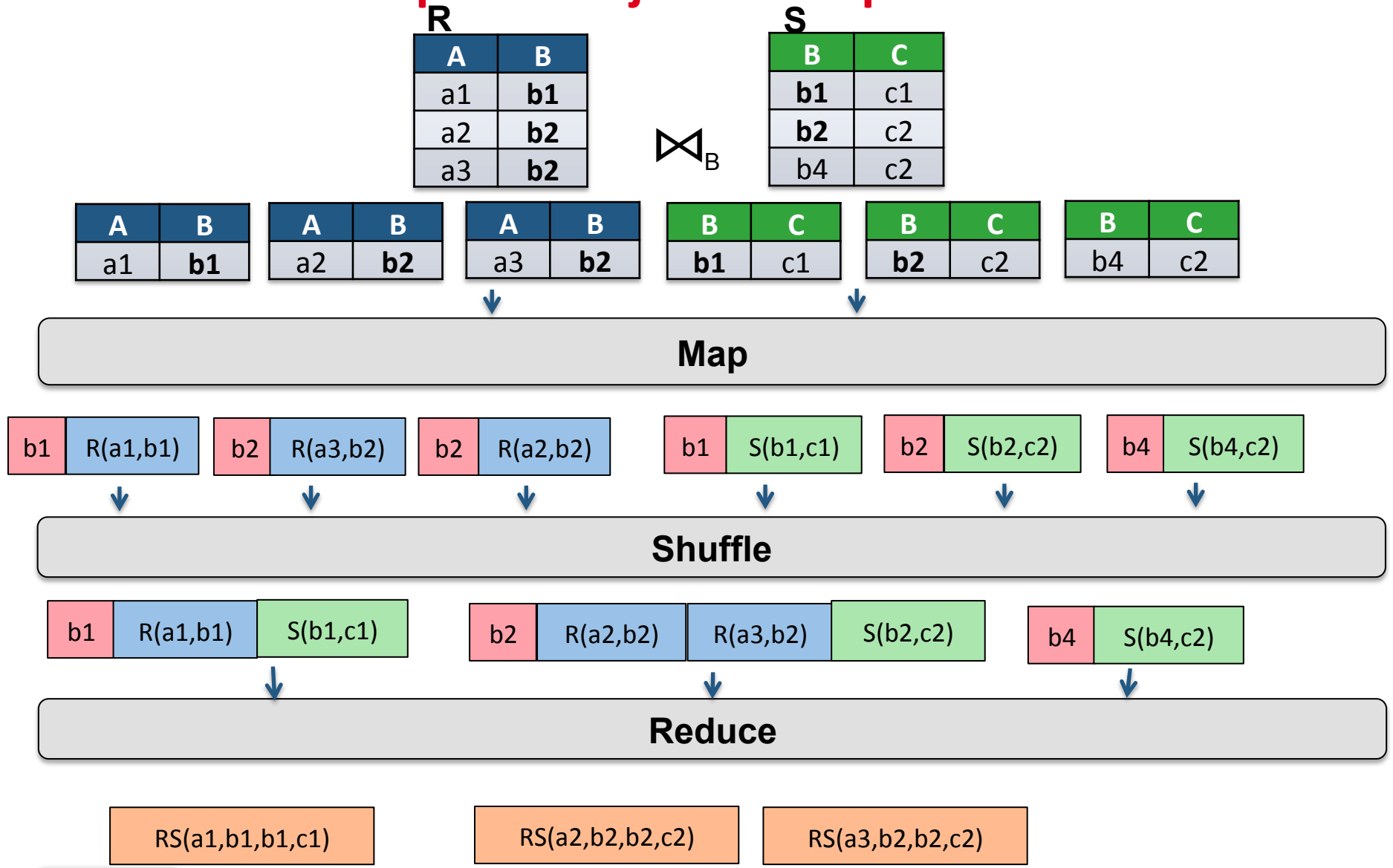
- ✚ High parallelization
- ✚ Scalable
  - number of nodes
  - size of data
- ✚ Fault-tolerant
  - replication of data
- ✚ Easy to implement

- ➖ Operates on only one input
- ➖ No indexes
  - scan of all data
- ➖ Blocking operations
  - e.g., reduce does not begin until all tasks have finished

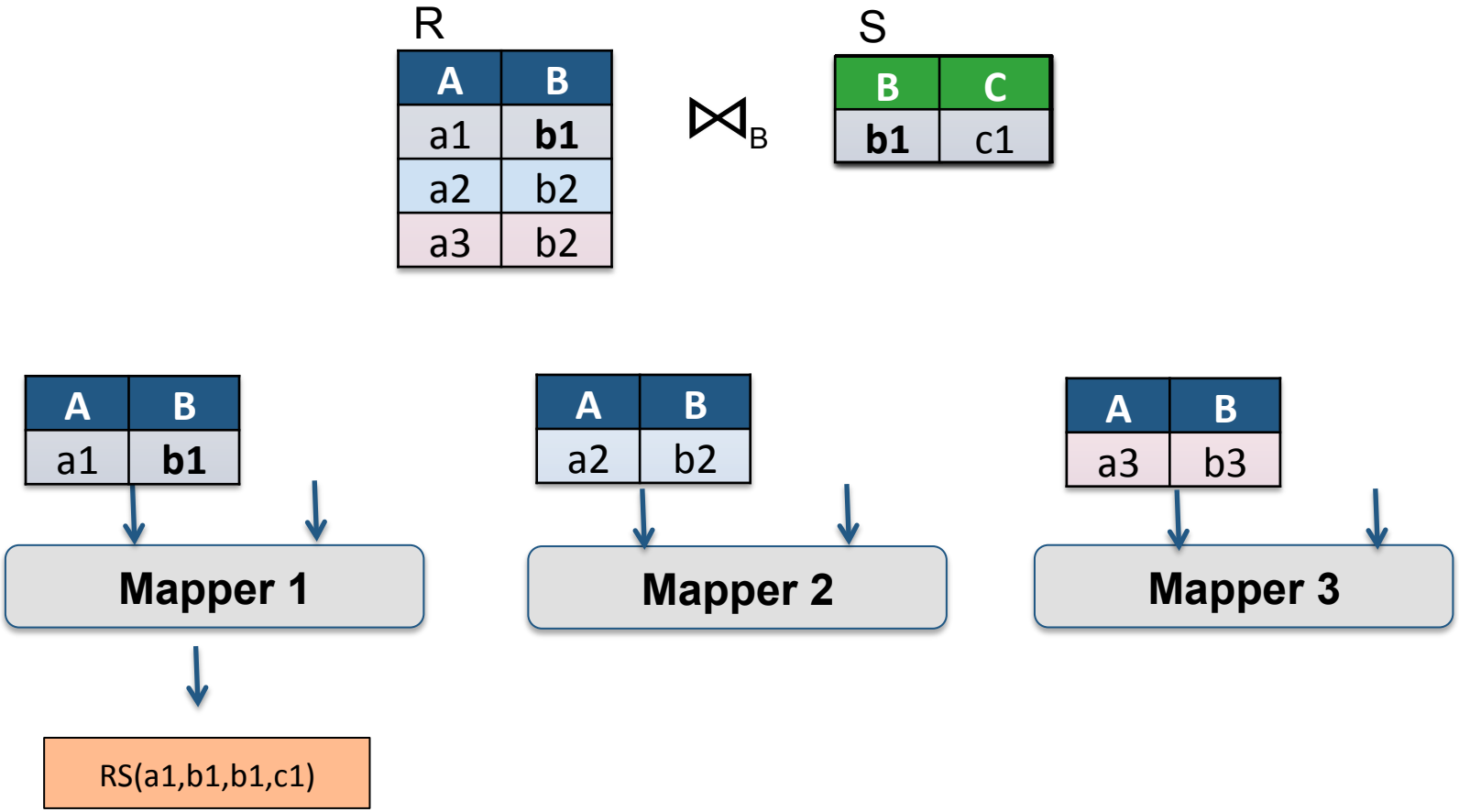
Joins in MapReduce

- Two main strategies:
  - symmetric hash join (repartitioned join)
  - replicated join (broadcast join)
- Representative works:
  - [Blanas10] on different join methods in MapReduce
  - [Afrati11] on optimizing multiway joins in MapReduce
  - [Wu11] on query optimization in MapReduce

Repartition join in MapReduce



Replicated (broadcast) join in MapReduce



Variability in the clouds

Cloud = performance variability [Schad10], [Iosup11]

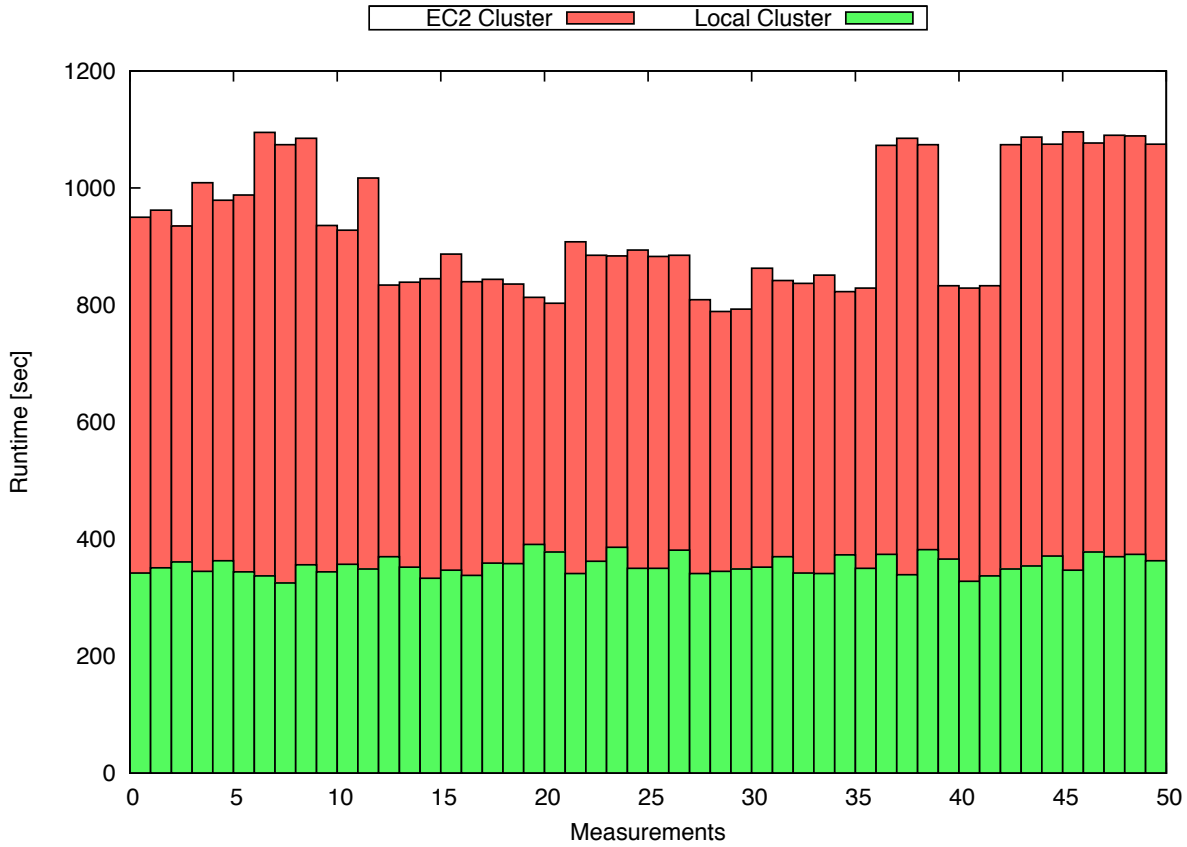


Figure 1: Runtime for a MapReduce job.

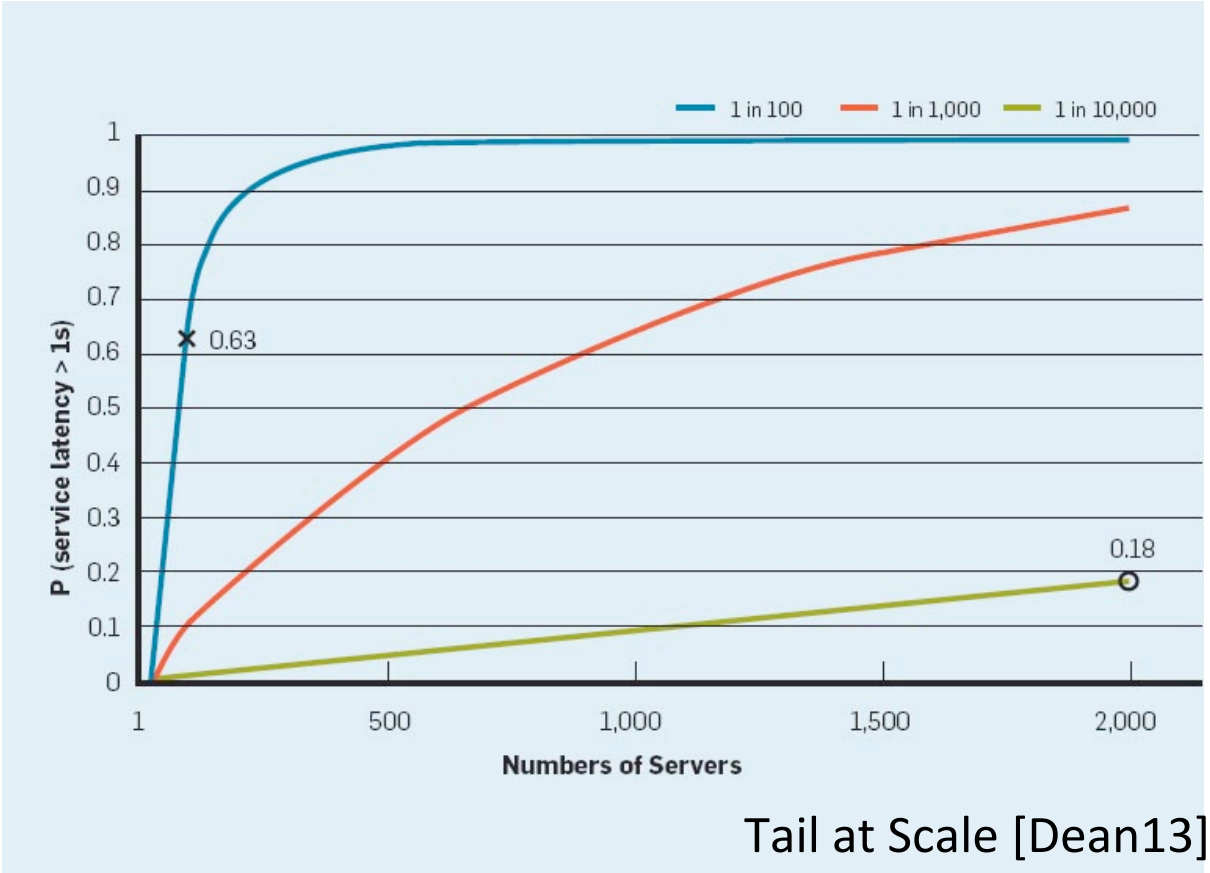
from [Schad10]

Why Variability Exists?

- Software-based reasons:
  - Shared resources (cpu, memory, network bandwidth)
  - Daemons running in the background
  - Global resource sharing such as network switches and shared file systems
  - Maintenance activities running in the background
  - Multiple layers of queuing
- Hardware-based reasons:
  - Garbage collection for SSD
  - Energy management

Variability in the clouds

Variability is amplified by scale



Tail at Scale [Dean13]

3

ANALYSIS OF RDF SYSTEMS IN THE CLOUD

RDF systems in the cloud

- Relatively new, fast-growing research area

- Numerous systems
  - **AMADA** [Bugiotti11, Arandar12]
  - CumulusRDF [Ladwig11]
  - EAGRE [Zhang13]
  - **Graph partitioning** [Huang11]
  - H2RDF [Papailiou12]
  - **HadoopRDF** [Husain11]
  - MAPSIN [Schätzle12]
  - PigSPARQL [Schätzle11]
  - *CliqueSquare (TBA…)*
  - RAPID+ [Ravindra11]
  - **Rya** [Punnoose12]
  - Stratustore [Ladwig11]
  - **SHARD** [Rohloff10]
  - **Trinity.RDF** [Zeng13]
  - **WebPie** [Urbani09]
  - QueryPie [Urbani11]
  - …

Analysis dimensions

1. Data storage

2. Query processing

3. RDFS entailment

Analysis dimensions

1. Data storage

2. Query processing

3. RDFS entailment

Categorization based on storage and query processing

I. Approaches based on MapReduce and DFS

II. Approaches based on key-value stores

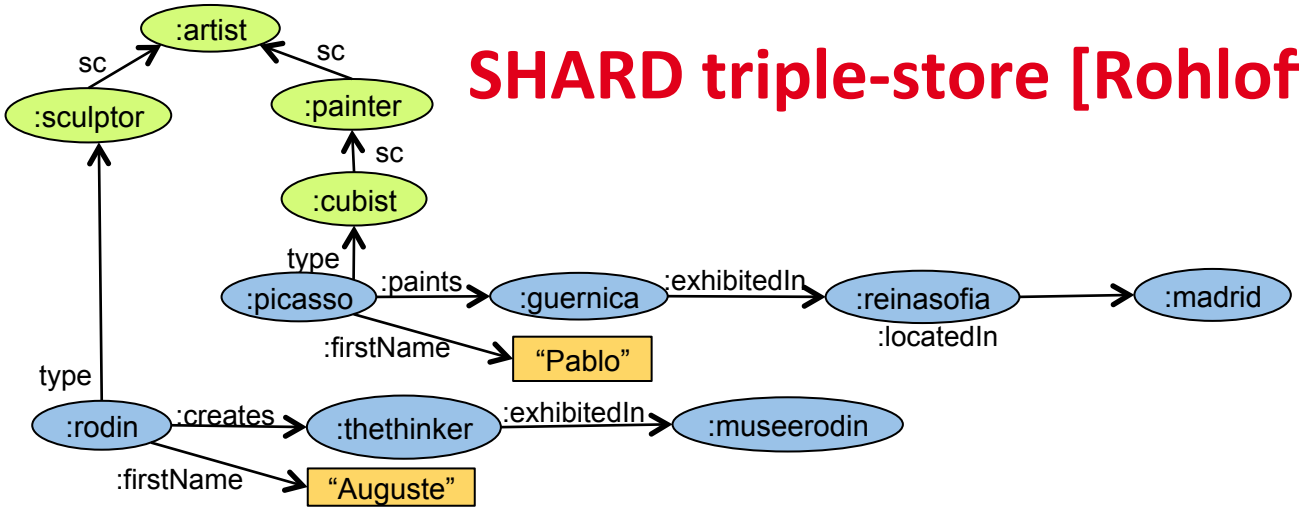
III. Graph-oriented approaches

IV. Systems based on commercial cloud services

I. MapReduce/DFS-based approaches

I. MapReduce/DFS-based approaches

- Store RDF triples in DFS (usually HDFS)
- Tranlate the query to MapReduce jobs
- Run MapReduce to answer the queries (usually Hadoop)
- Representative works:
  - SHARD [Rohloff10]
  - HadoopRDF [Husain11]
  - RAPID+ [Ravindra11]
  - PigSPARQL [Schätzle11]



SHARD triple-store [Rohloff10]

Storing: each line holds all triples about a given subject

File1.rdf

```
picasso type :cubist :paints :guernica :firstName "Pablo"
guernica :exhibitedIn :reinasofia
reinasofia :locatedIn :madrid
rodin type :sculptor :firstName "Auguste" :creates :thethinker
thethinker :exhibitedIn :museerodin
```

File2.rdf

```
:sculptor sc :artist
:cubist sc :painter
:painter sc :artist
```

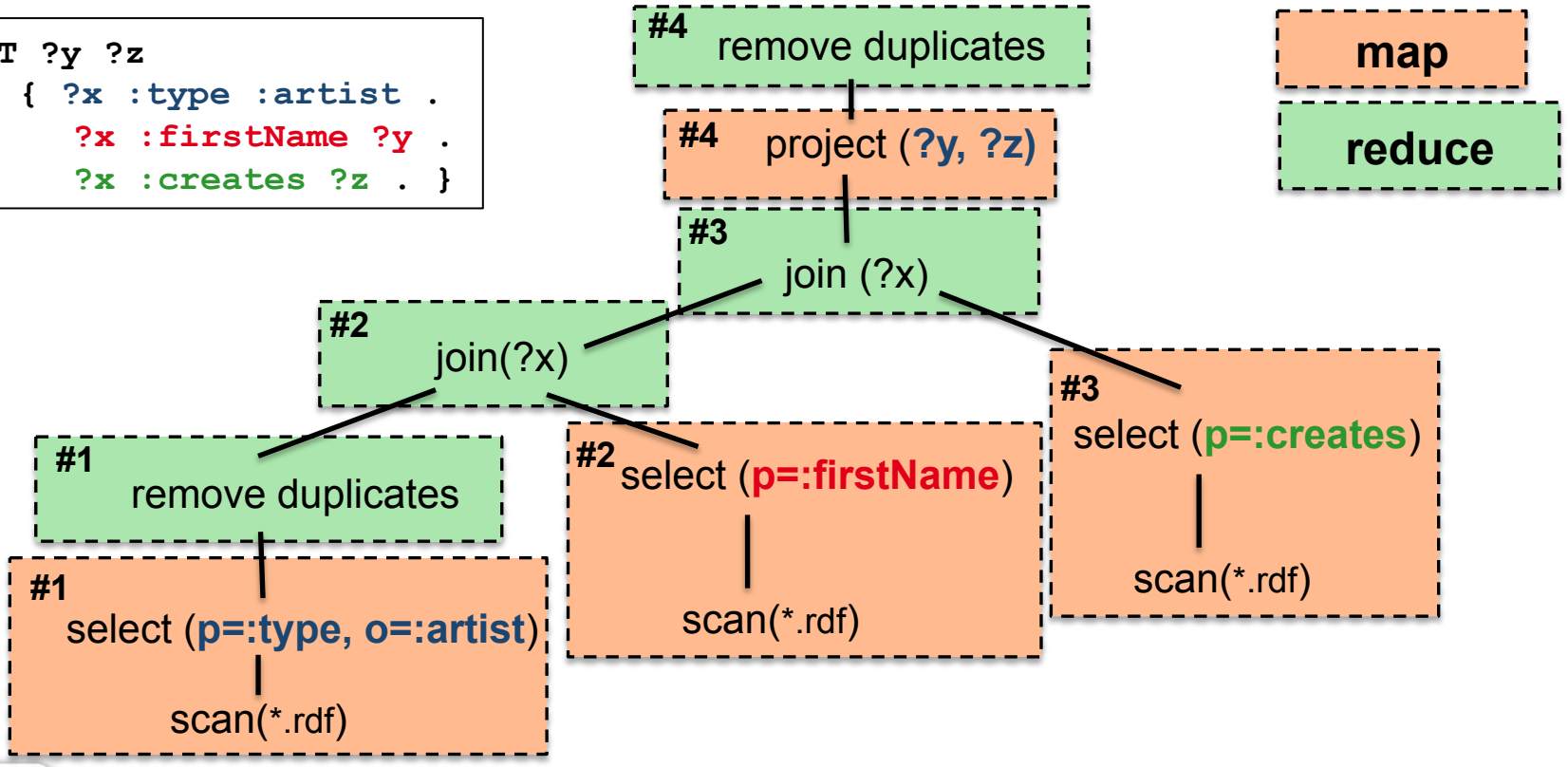
HDFS

SHARD triple-store [Rohloff10]

Query processing: Iterative MapReduce jobs

- 1 MapReduce job for each triple pattern + 1 for the projection at the end
- Map phase: match triple pattern by scanning all triples
- Reduce phase: join triple pattern with intermediate results from previous ones

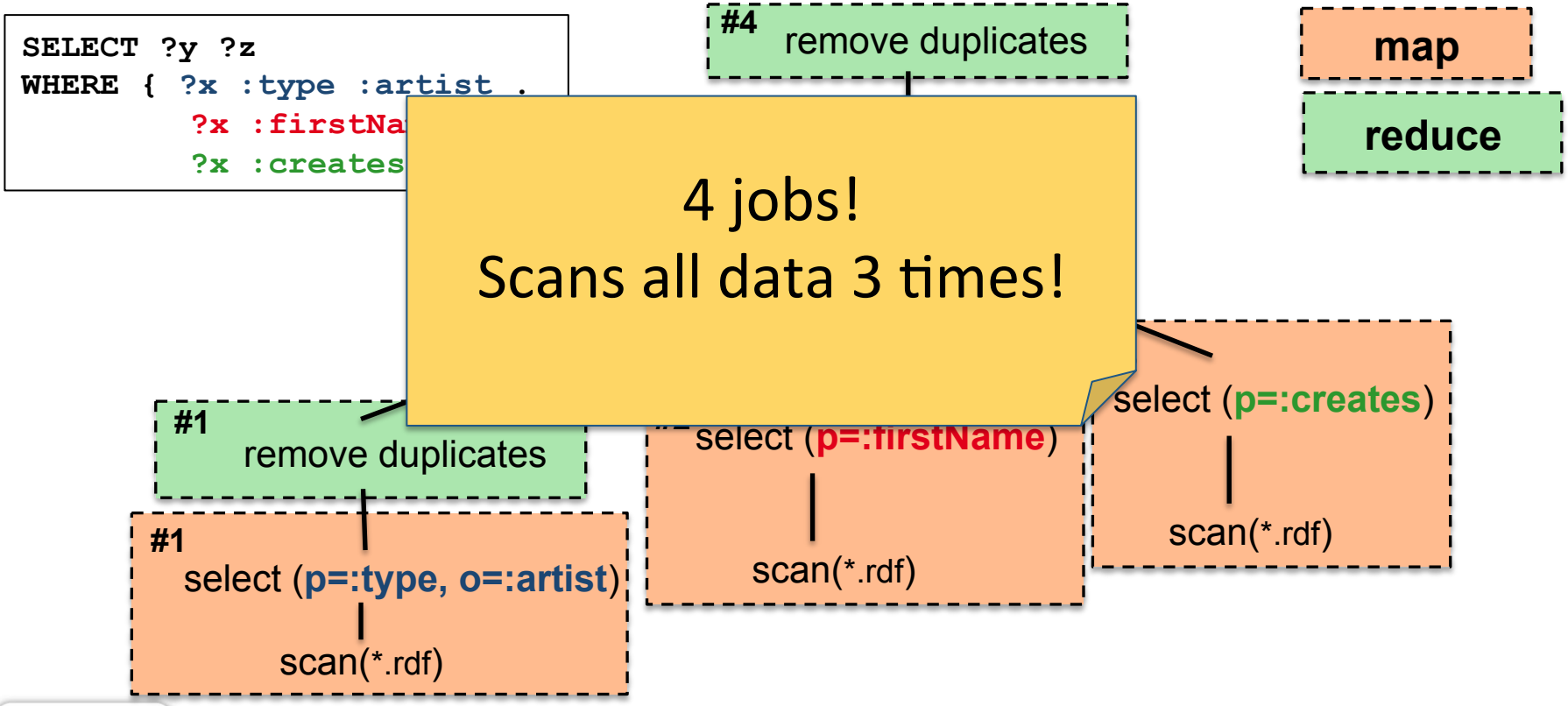
```
SELECT ?y ?z
WHERE { ?x :type :artist .
        ?x :firstName ?y .
        ?x :creates ?z . }
```



SHARD triple-store [Rohloff10]

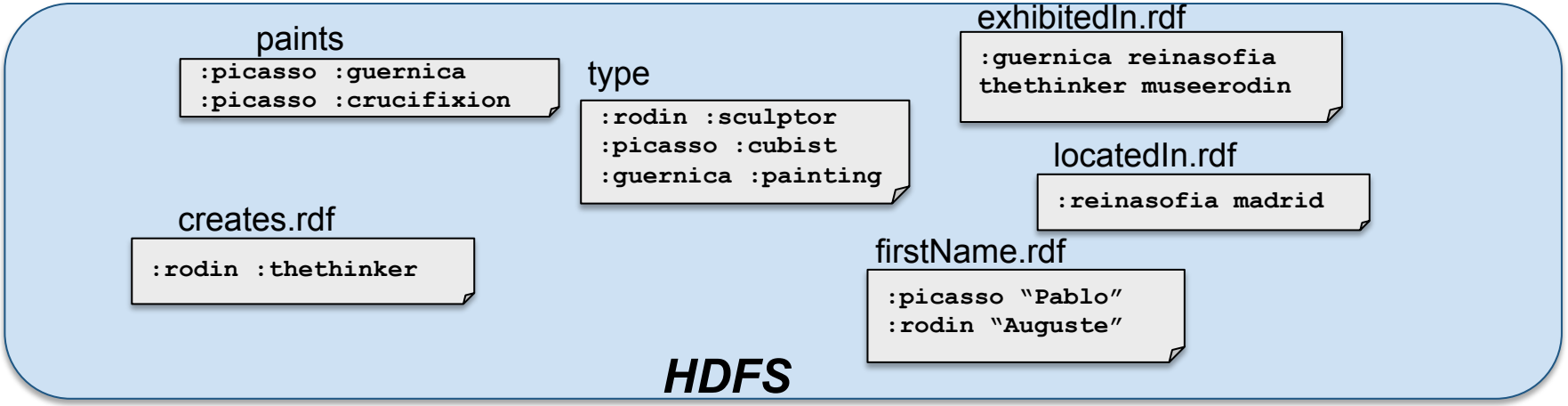
Query processing: Iterative MapReduce jobs

- 1 MapReduce job for each triple pattern + 1 for the projection at the end
- Map phase: match triple pattern by scanning all triples
- Reduce phase: join triple pattern with intermediate results from previous ones



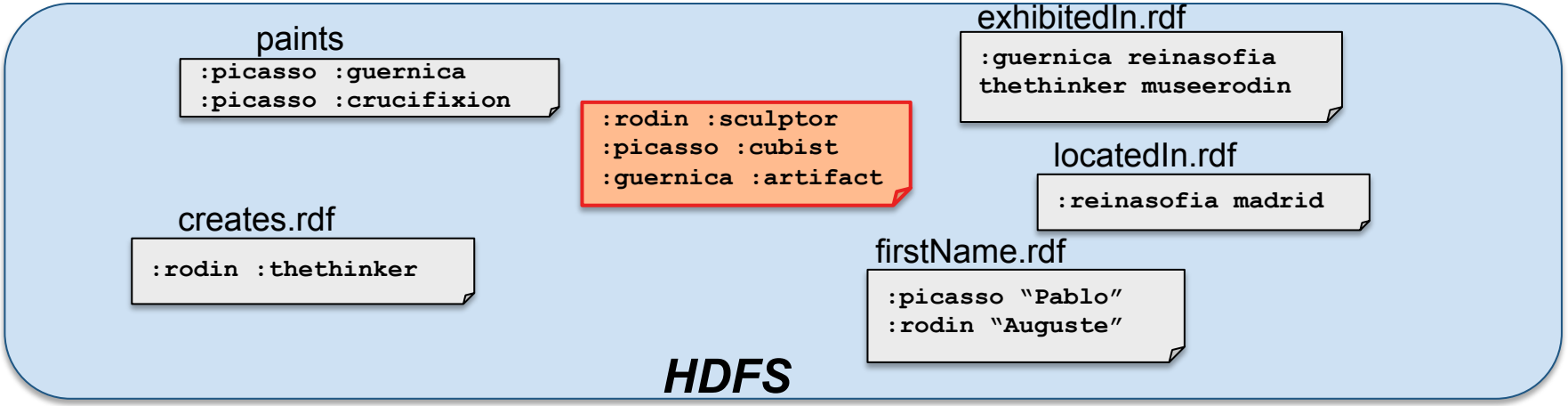
HadoopRDF - storage [Husain11]

- RDF triples are grouped by the property names in files (like in vertical partitioning of centralized RDF stores)



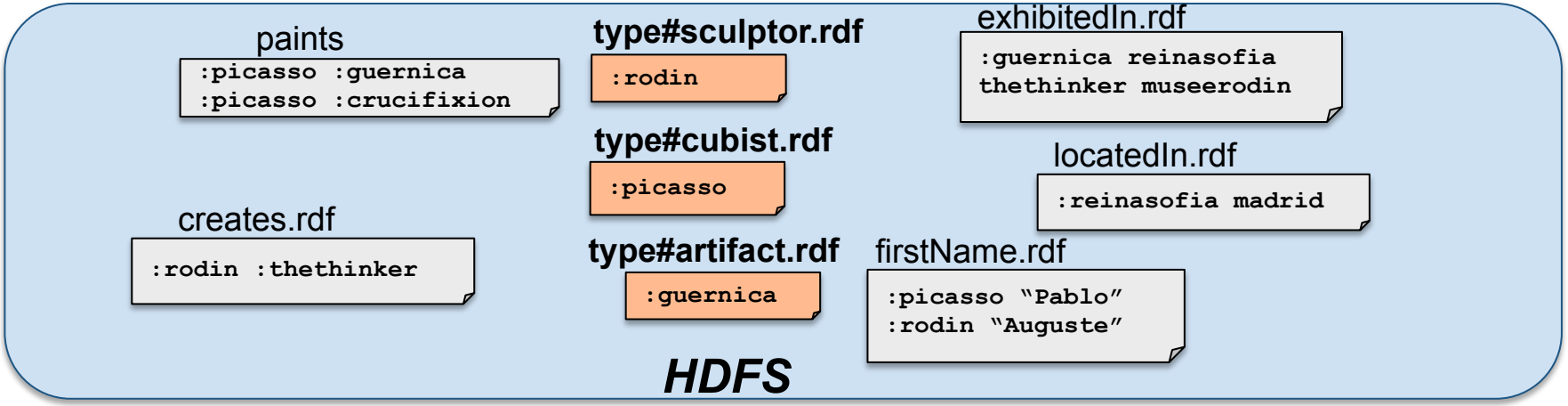
HadoopRDF - storage

- RDF triples are grouped by the property names in files (like in vertical partitioning of centralized RDF stores)
- The file for the property type is partitioned based on the object value



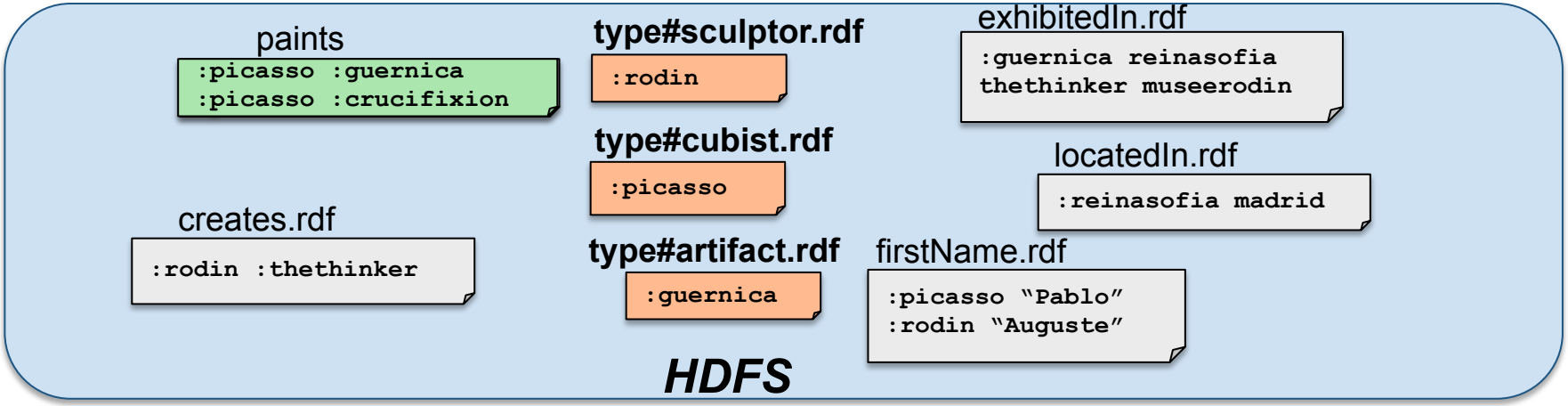
HadoopRDF - storage

- RDF triples are grouped by the property names in files (like in vertical partitioning of centralized RDF stores)
- The file for the property type is partitioned based on the object value



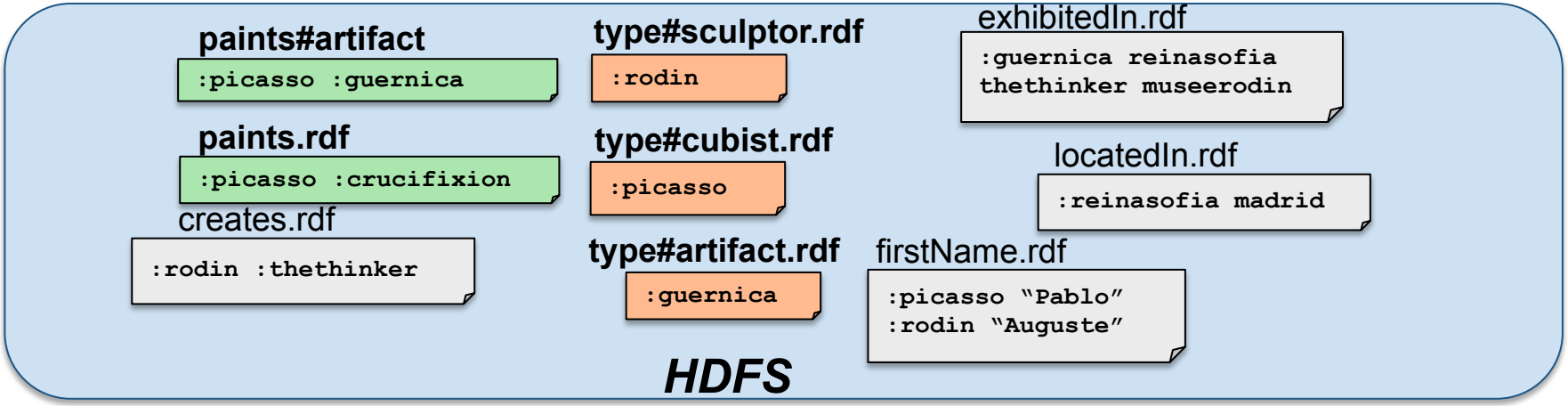
HadoopRDF - storage

- RDF triples are grouped by the property names in files (like in vertical partitioning of centralized RDF stores)
- The file for the property type is partitioned based on the object value
- All other property files are partitioned by their object type (if there is such information)



HadoopRDF - storage

- RDF triples are grouped by the property names in files (like in vertical partitioning of centralized RDF stores)
- The file for the property type is partitioned based on the object value
- All other property files are partitioned by their object type (if there is such information)



What do we gain from this storage scheme?

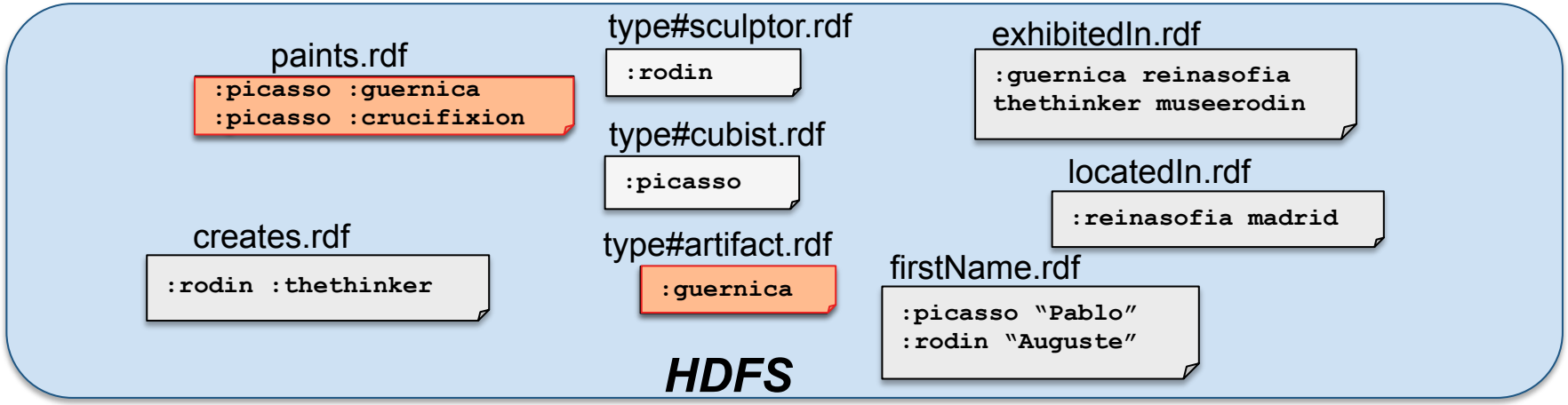
- Space gains

Step	Files	Size (GB)	Space Gain
N-Triples	20020	24	-
PS	17	7.1	70.42%
POS	41	6.6	7.04%

- Query optimizaton

```
SELECT ?x
WHERE { ?x :paints ?y .
        ?y type :artifact .}
```

With PS we would have to scan files: {paints.rdf, type#artifact.rdf)



What do we gain from this storage scheme?

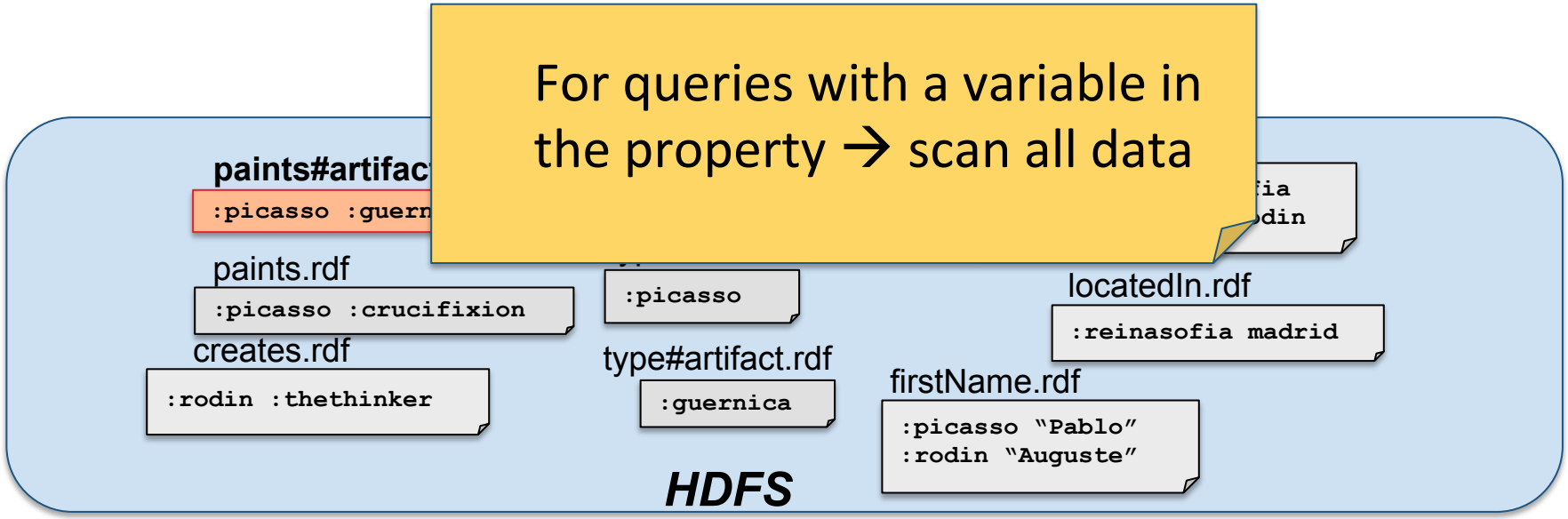
- Space gains

Step	Files	Size (GB)	Space Gain
N-Triples	20020	24	-
PS	17	7.1	70.42%
POS	41	6.6	7.04%

- Query optimizaton

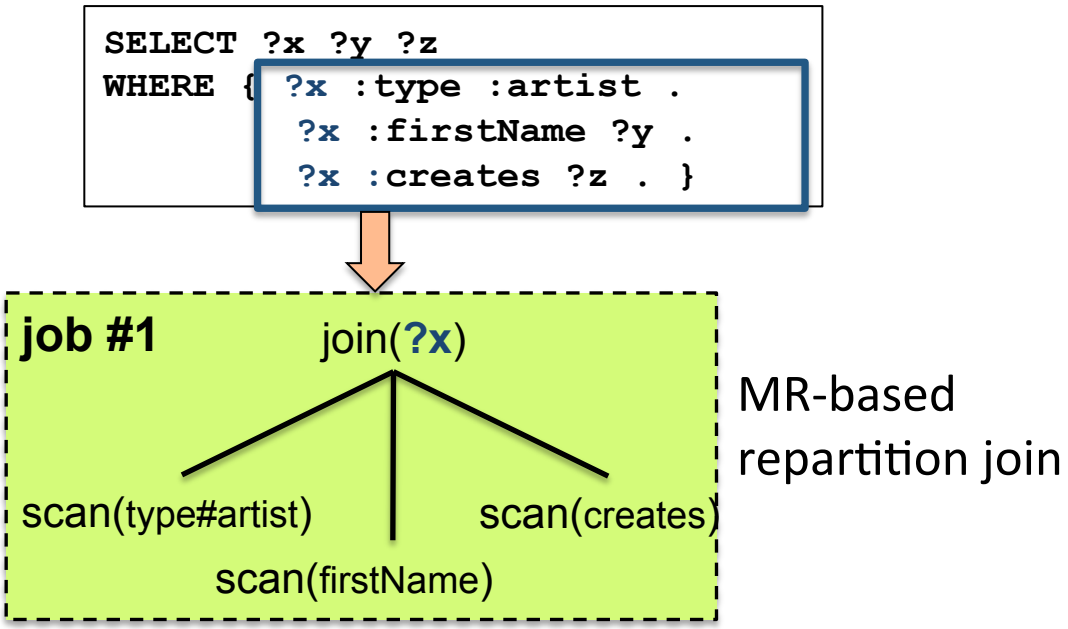
```
SELECT ?x
WHERE { ?x :paints ?y .
        ?y type :artifact .}
```

But with POS we only need to scan: {paints#artifact.rdf)



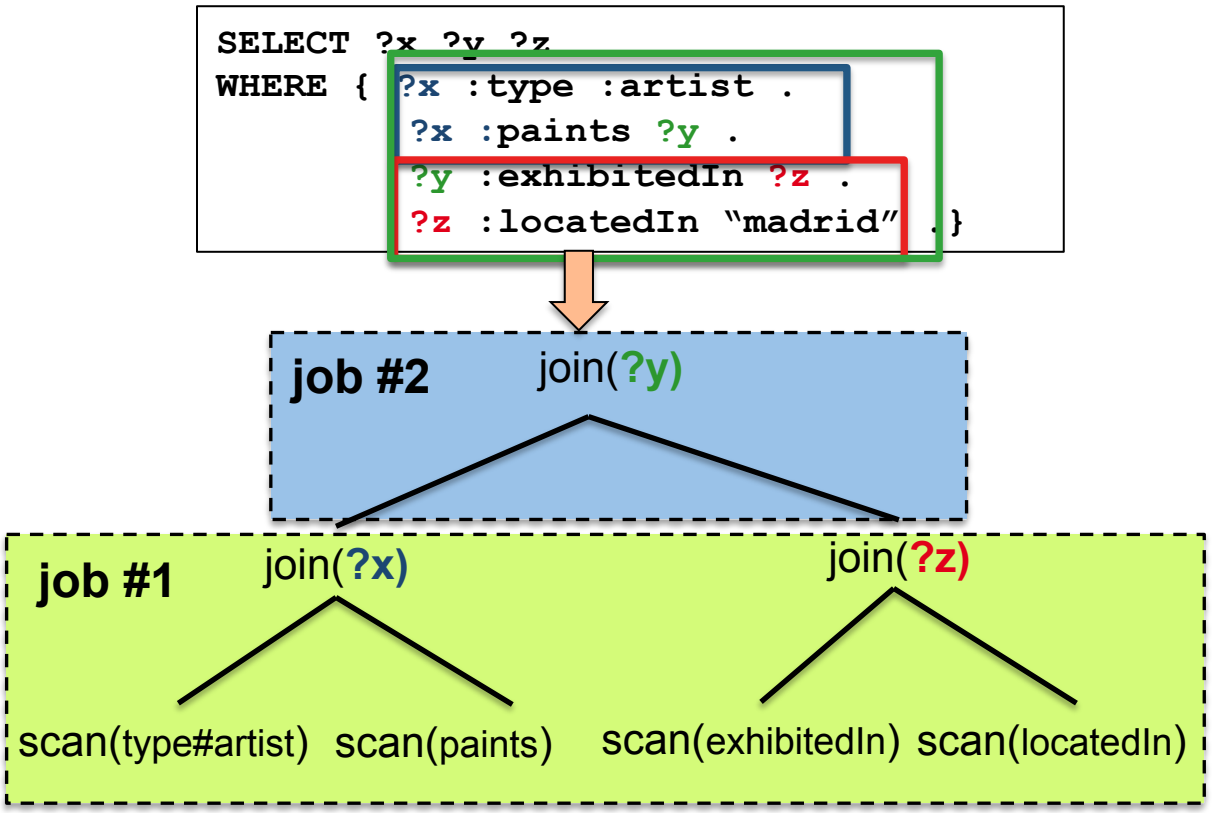
HadoopRDF - querying

- File selection based on the property (and object) of the triple
- Joining two or more triple patterns on the same variable



HadoopRDF - querying

- File selection based on the property (and object) of the triple
- Joining two or more triple patterns on the same variable
- Several joins may be executed in parallel if they are on different variables



MR-based repartition joins

HadoopRDF - querying

How is a query decomposed? What query plan to create?

- Observations
  - Significant overhead when running a job
  - Traditional selectivity-based optimization techniques may result to plans with more jobs → bigger response times

Dataset	2 Job Plan	3 Job Plan	Difference
LUBM_10000	4920	9180	4260
LUBM_20000	31020	36540	5520
LUBM_30000	80460	93947	13487

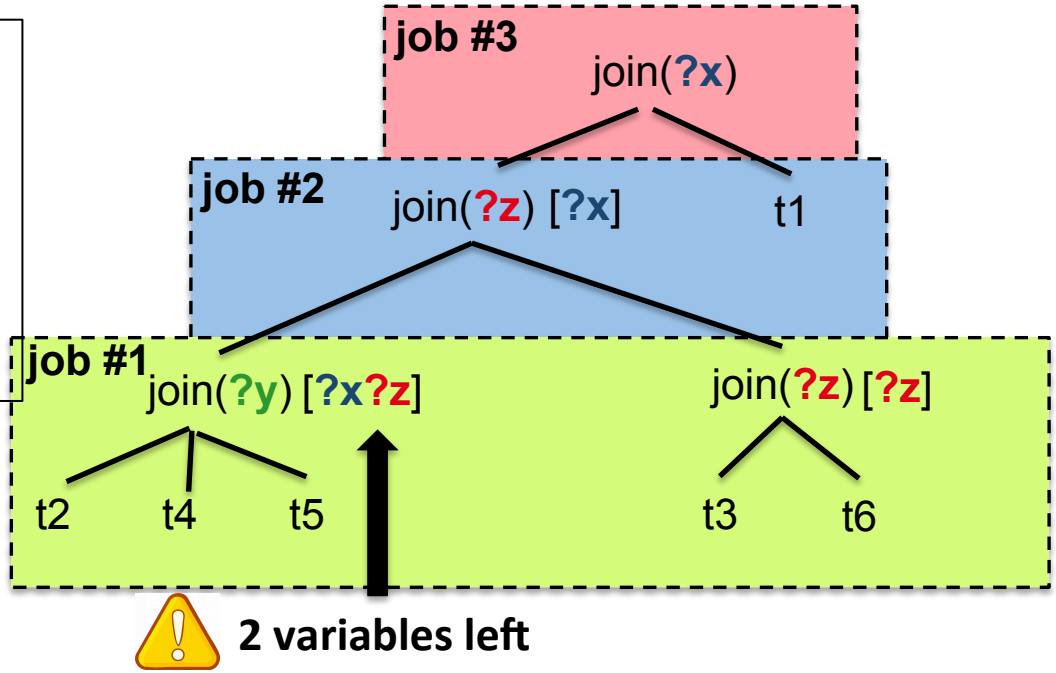
- Heuristic: Query decomposition to minimize the number of Hadoop jobs

HadoopRDF – query planning

- Greedy algorithm
- As many joins as possible in one job
- Each triple pattern is used only once in each join
- Leave the least number of remaining variables to be joined

```
SELECT ?x ?y ?z
WHERE {?x type :artist .          (t1)
       ?y type :artifact .        (t2)
       ?z type :museum .          (t3)
       ?x :paints ?y .            (t4)
       ?y :exhibitedIn ?z .       (t5)
       ?z :locatedIn "madrid"     (t6)
}
```

Complete variable elimination of ?y

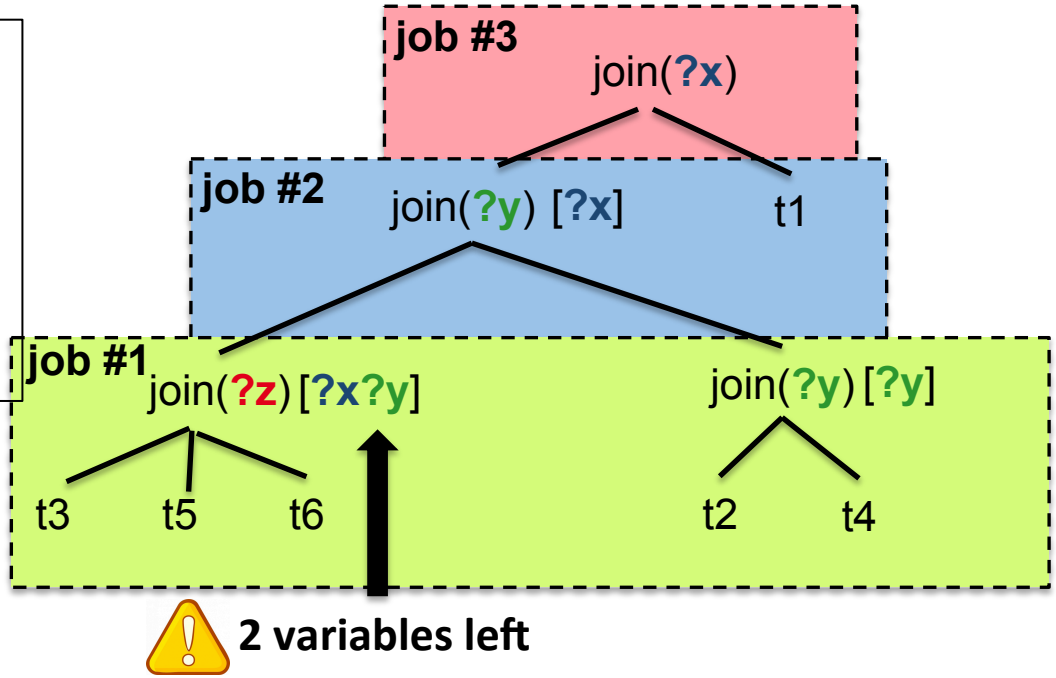


HadoopRDF – query planning

- Greedy algorithm
- As many joins as possible in one job
- Each triple pattern is used only once in each join
- Leave the least number of remaining variables to be joined

```
SELECT ?x ?y ?z
WHERE {?x type :artist .          (t1)
       ?y type :artifact .        (t2)
       ?z type :museum .          (t3)
       ?x :paints ?y .            (t4)
       ?y :exhibitedIn ?z .       (t5)
       ?z :locatedIn "madrid"     (t6)
}
```

Complete variable elimination of ?z



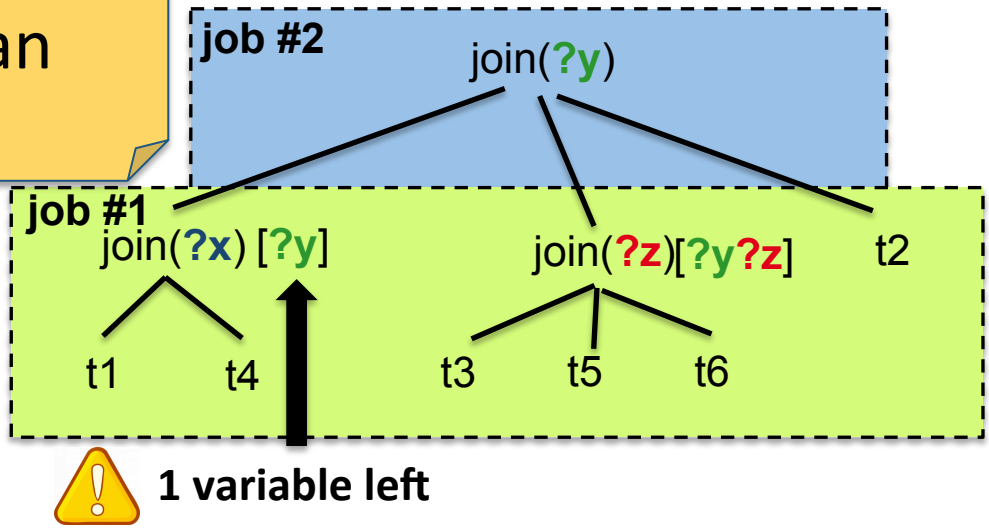
HadoopRDF – query planning

- Greedy algorithm
- As many joins as possible in one job
- Each triple pattern is used only once in each join
- Leave the least number of remaining variables to be joined

```
SELECT ?x ?y ?z
WHERE {?x type :artist
       ?y type :artifa
       ?z type :museum
       ?x :paints ?y .
       ?y :exhibitedIn ?z .   (t5)
       ?z :locatedIn "madrid" (t6)
}
```

Best plan

Complete variable elimination of ?x



HadoopRDF - results

	LUBM-10,000		LUBM-20,000		LUBM-30,000	
	RDF-3X	HadoopRDF	RDF-3X	HadoopRDF	RDF-3X	HadoopRDF
Query 1	0.373	248.3	0.219	418.523	0.412	591
Query 2	1240.21	801.9	3518.485	924.437	**FAILED**	1315.2
Query 4	0.856	1430.2	0.445	2259.768	1.160	3405.6
Query 9	225.54	1663.4	**FAILED**	3206.439	**FAILED**	4693.8
Query 12	0.298	204.4	0.825	307.715	1.254	425.4
Query 13	380.731	325.4	480.758	462.307	1287	697.2

I. MapReduce/DFS-based approaches

Pros	Cons
Scalability	No indexes – scan all data
Fault-tolerance	Shuffling of data
Fast data uploading	Semi-support for joins
Fast and easy to implement	Big initialization overhead
Good for “analytical” queries	High query response times

II. Approaches based on key-value stores

II. Approaches based on key-value stores

- Store RDF triples in a key-value store
  - Indexes on RDF triples
- Lookups to the key-value store for matching each triple pattern
- How do I perform the joins?
  - Collect the results somewhere locally
  - Use MapReduce
- Representative works
  - CumulusRDF [Ladwig11]
  - Stratustore [Stein10]
  - Rya [Punnoose12]
  - AMADA [Bugiotti12]
  - H2RDF [Papailiou12]
  - MAPSIN [Schätzle12]

Local query processing

MapReduce-based query processing

Storing RDF in key-value stores

- Rya uses Apache Accumulo — **Sorted Index**
  - SPO|-|- POS|-|- OSP|-|-
- H2RDF uses Apache HBase — **Sorted Index**
  - SP|O|- PO|S|- OS|P|-
- AMADA uses Amazon DynamoDB — **Hash Index**
  - S|P|O P|O|S O|S|P
- MAPSIN uses Apache HBase — **Sorted Index**
  - S|P|O|- O|P|S|-
- Stratustore uses Amazon SimpleDB — **Hash Index**
  - S|P|O
- CumulusRDF uses Apache Cassandra — **Hash Index** **Sorted Index**
  - hierarchical layout: S|{P}O|- P|{O}S|- O|{S}P|-
  - flat layout: S|PO|- PO|S|- PO|“P”| P O|SP|-

Storing RDF in key-value stores

- Rya uses Apache Accumulo — **Sorted Index**
  - SPO|-|- POS|-|- OSP|-|-
- H2RDF uses Apache HBase — **Sorted Index**
  - SP|O|- PO|S|- OS|P|-
- AMADA uses Am… — …dex
  - S|P|O P|…
- MAPSIN uses Apa… — …ndex
  - S|P|O|- O|…
- Stratustore uses Amazon SimpleDB — **Hash Index**
  - S|P|O
- CumulusRDF uses Apache Cassandra — **Hash Index** **Sorted Index**
  - hierarchical layout: S|{P}O|- P|{O}S|- O|{S}P|-
  - flat layout: S|PO|- PO|S|- PO|"P"| P O|SP|-

3 indexes:
SPO, POS, OSP

Rya storage - example

SPO

Key	(attribute, value)
:picasso,:firstName,"Pablo"	-
:picasso,:p	
:picasso,ty	
:guernica,:	
…	

POS

Key	(attribute, value)
:exhibitedIn,:reinasofia,:guernica	-
:firstName,"Pablo",:picasso,	-
:paints,:gu	
type,:cubis	
…	

OSP

Key	(attribute, value)
:cubist,:picasso,type,	-
:guernica,:picasso,:paints	-
"Pablo",:picasso,:firstName	-
:reinasofia,:guernica,:exhibitedIn	-
…	-

Strong point of key-value stores: RDF lookups

Different access paths for triple pattern matching

Triple pattern	Rya (Accumulo)	H2RDF (HBase)	AMADA (DynamoDB)	MAPSIN (HBase)
(s, p, o)	any lookup	any lookup + select	any lookup + select	any lookup + sselect
(s, p, ?o)	SPO range scan	SP|O lookup	S|P|O lookup + select	S|P|O lookup + sselect
(s, ?p, o)	OSP range scan	OS|P lookup	O|S|P lookup + select	O|S|P lookup + sselect
(s, ?p, ?o)	SPO range scan	SP|O range scan	S|P|O lookup	S|P|O lookup
(?s, p, o)	POS range scan	PO|S lookup	P|O|S lookup + select	O|S|P lookup + sselect
(?s, p, ?o)	POS range scan	PO|S range scan	P|O|S lookup	any scan + sselect
(?s, ?p, o)	OSP range scan	OS|P range scan	O|S|P lookup	O|S|P lookup
(?s, ?p, ?o)	any scan	any scan	any scan	any scan

select: filter on client side
sselect: filter on server side

Weak point of key-value stores: joins

Key-value stores do not support joins

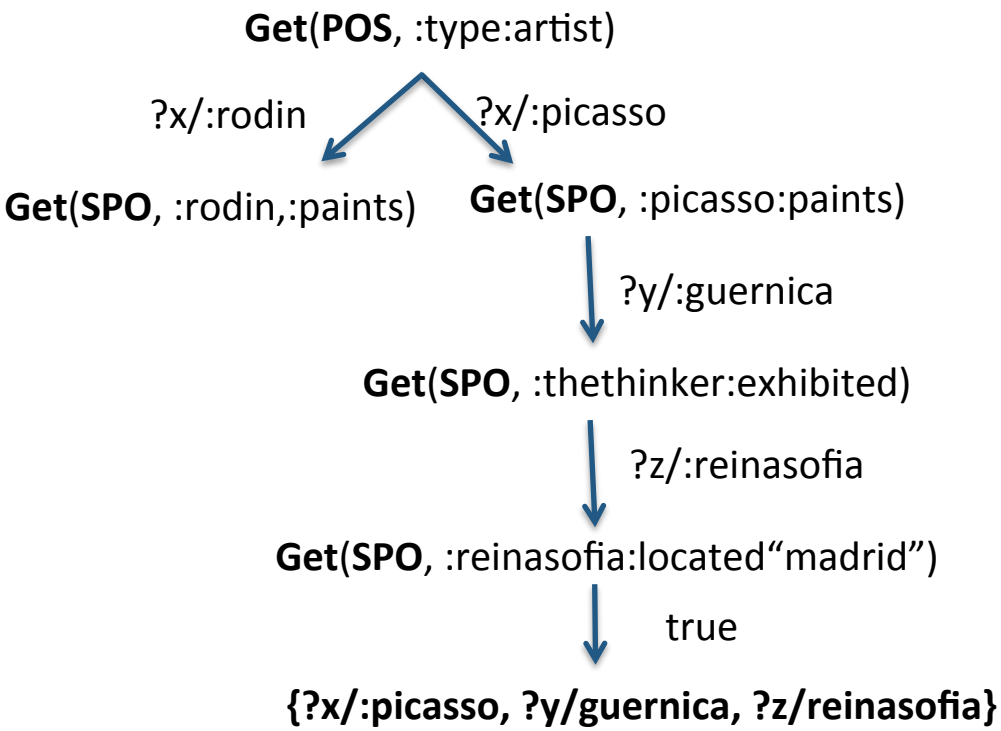
Make all lookups and perform joins out of the store

- **Rya**: index nested loops by query rewriting and lookups
- **H2RDF**: 2 strategies
  - selective joins → centralized – index nested loops
  - non-selective → MapReduce
- **AMADA**: memory hash join
- **MAPSIN**: MapReduce (map phase only)
- **Stratustore**: SimpleDB SELECT query for each star-join and rest of the joins locally
- **CumulusRDF**: Only single triple pattern queries are supported

Rya joining strategy

- Index nested loops
- A lookup for the 1st triple pattern
- For each triple pattern t_i, use the n results from triple pattern t_{i-1} to rewrite t_i and then perform n lookups for the rewritten t_i

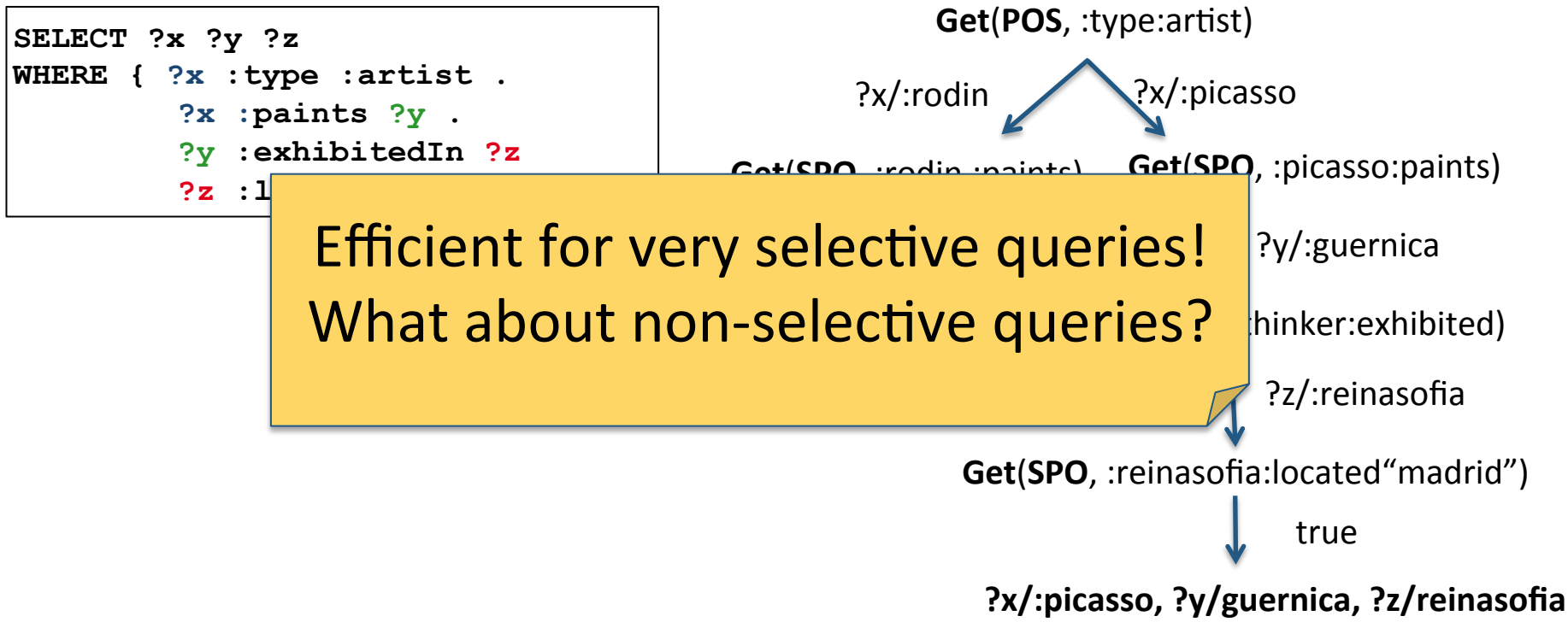
```
SELECT ?x ?y ?z
WHERE { ?x :type :artist .
        ?x :paints ?y .
        ?y :exhibitedIn ?z
        ?z :locatedIn "madrid".}
```



Rya joining strategy

- Index nested loops
- A lookup for the 1st triple pattern
- For each triple pattern t_i, use the n results from triple pattern t_{i-1} to rewrite t_i and then perform n lookups for the rewritten t_i

```
SELECT ?x ?y ?z
WHERE { ?x :type :artist .
        ?x :paints ?y .
        ?y :exhibitedIn ?z
        ?z :l
```



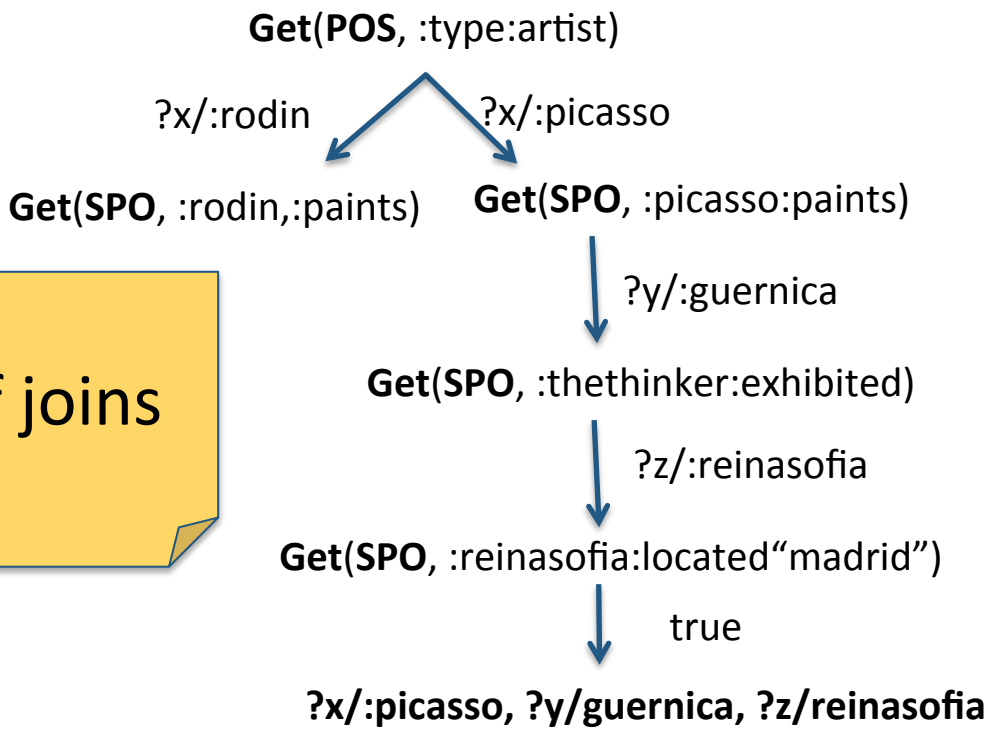
Efficient for very selective queries!
What about non-selective queries?

Rya - optimizations

- Index nested loops
- A lookup for the 1st triple pattern
- For each triple pattern t_i, use the n results from triple pattern t_{i-1} to rewrite t_i and then perform n lookups for the rewritten t_i

```
SELECT ?x ?y ?z
WHERE { ?x :type :artist .
        ?x :paints ?y .
        ?y :exhibitedIn ?z
        ?z :locatedIn "madrid".}
```

1. Parallelization of joins

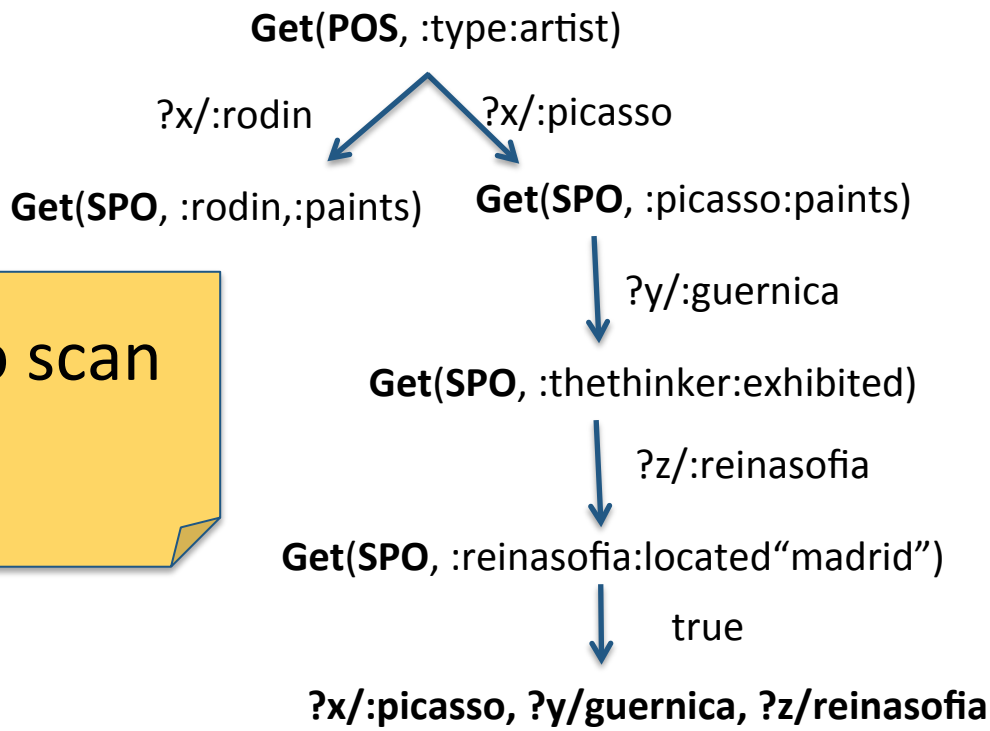


Rya - optimizations

- Index nested loops
- A lookup for the 1st triple pattern
- For each triple pattern t_i, use the n results from triple pattern t_{i-1} to rewrite t_i and then perform n lookups for the rewritten t_i

```
SELECT ?x ?y ?z
WHERE { ?x :type :artist .
        ?x :paints ?y .
        ?y :exhibitedIn ?z
        ?z :locatedIn "madrid".}
```

2. Batch scanner to scan many ranges

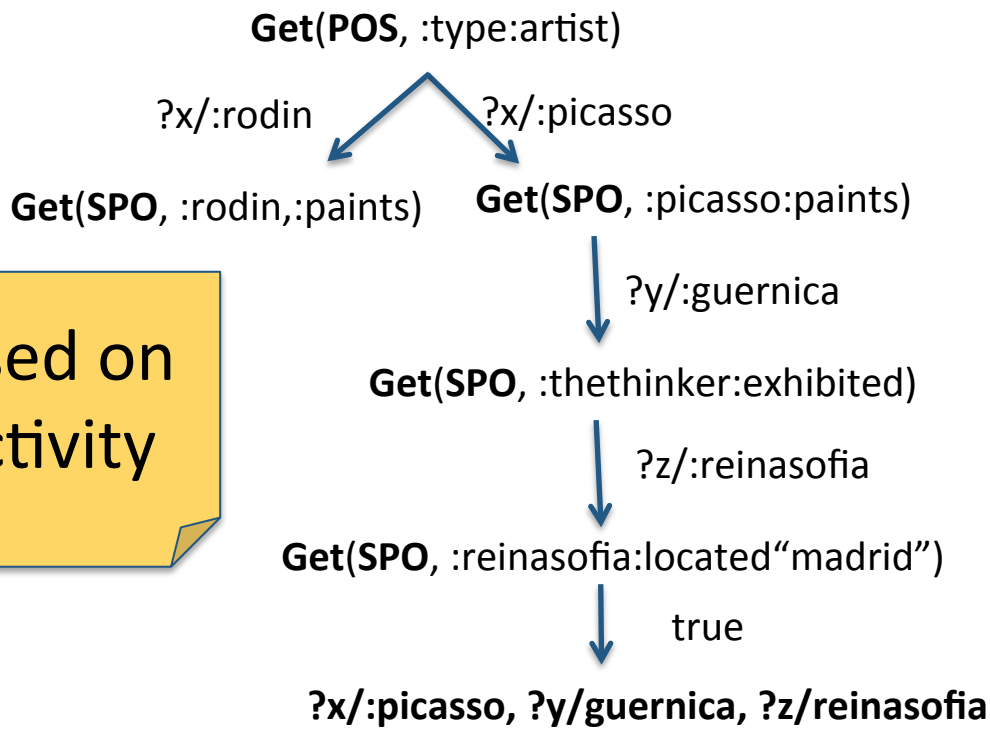


Rya - optimizations

- Index nested loops
- A lookup for the 1st triple pattern
- For each triple pattern t_i, use the n results from triple pattern t_{i-1} to rewrite t_i and then perform n lookups for the rewritten t_i

```
SELECT ?x ?y ?z
WHERE { ?x :type :artist .
        ?x :paints ?y .
        ?y :exhibitedIn ?z
        ?z :locatedIn "madrid".}
```

3. Join ordering based on triple pattern selectivity



II. Approaches based on key-value stores

Pros	Cons
Scalability	Low schema expressivity
Fault-tolerance	Joins implemented by the user
Very fast lookups	Joins on the client-side
Efficient for selective queries	

III. Graph-oriented approaches

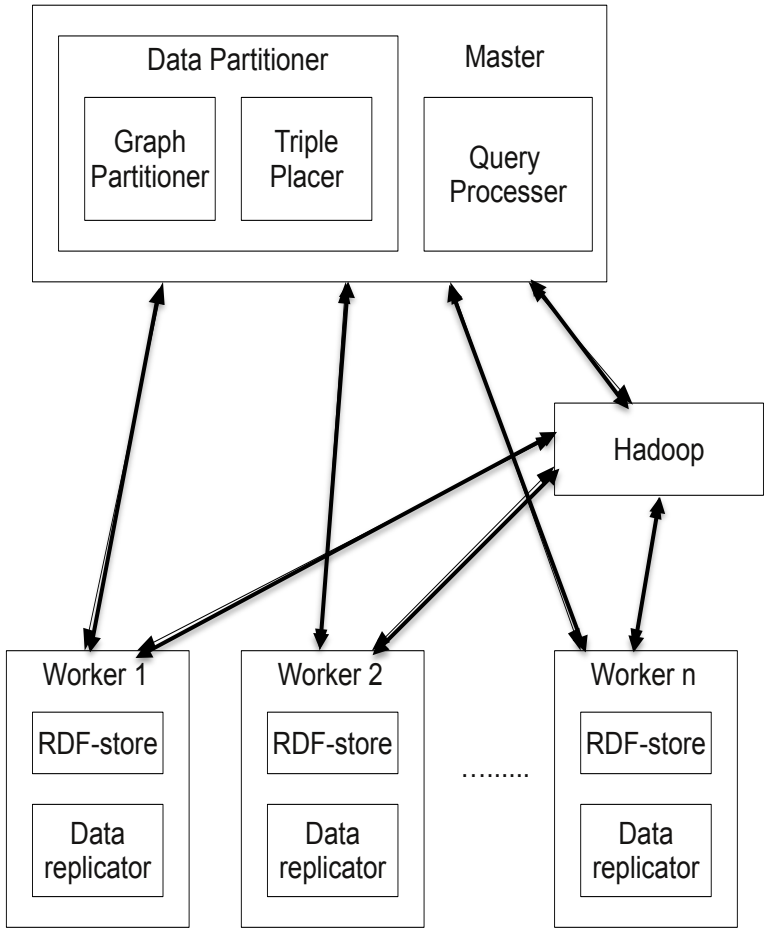
III. Graph-oriented approaches

- RDF data partitioning using known graph tools
  - [Huang11] uses METIS: a graph partitioning tool
  - [Hose13] extends [Huang11] for known query workloads

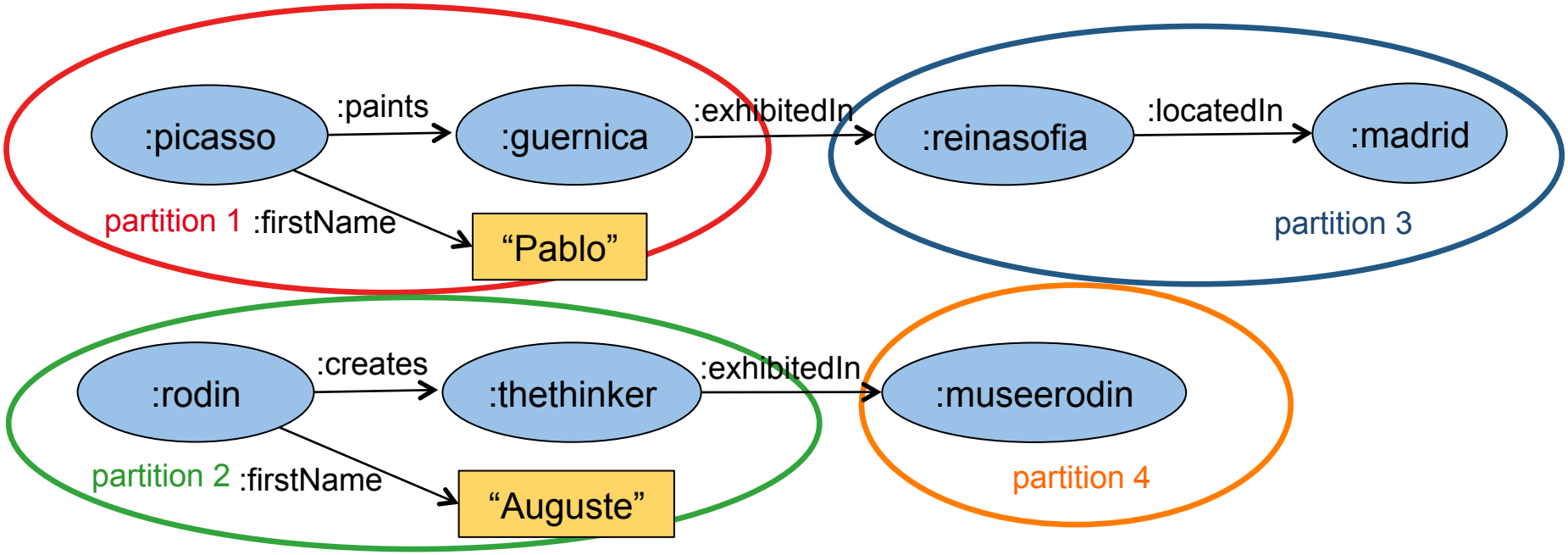
- Trinity.RDF [Zeng13] by Microsoft
  - Uses Trinity: a distributed graph system over a memory cloud

Graph partitioning [Huang11]

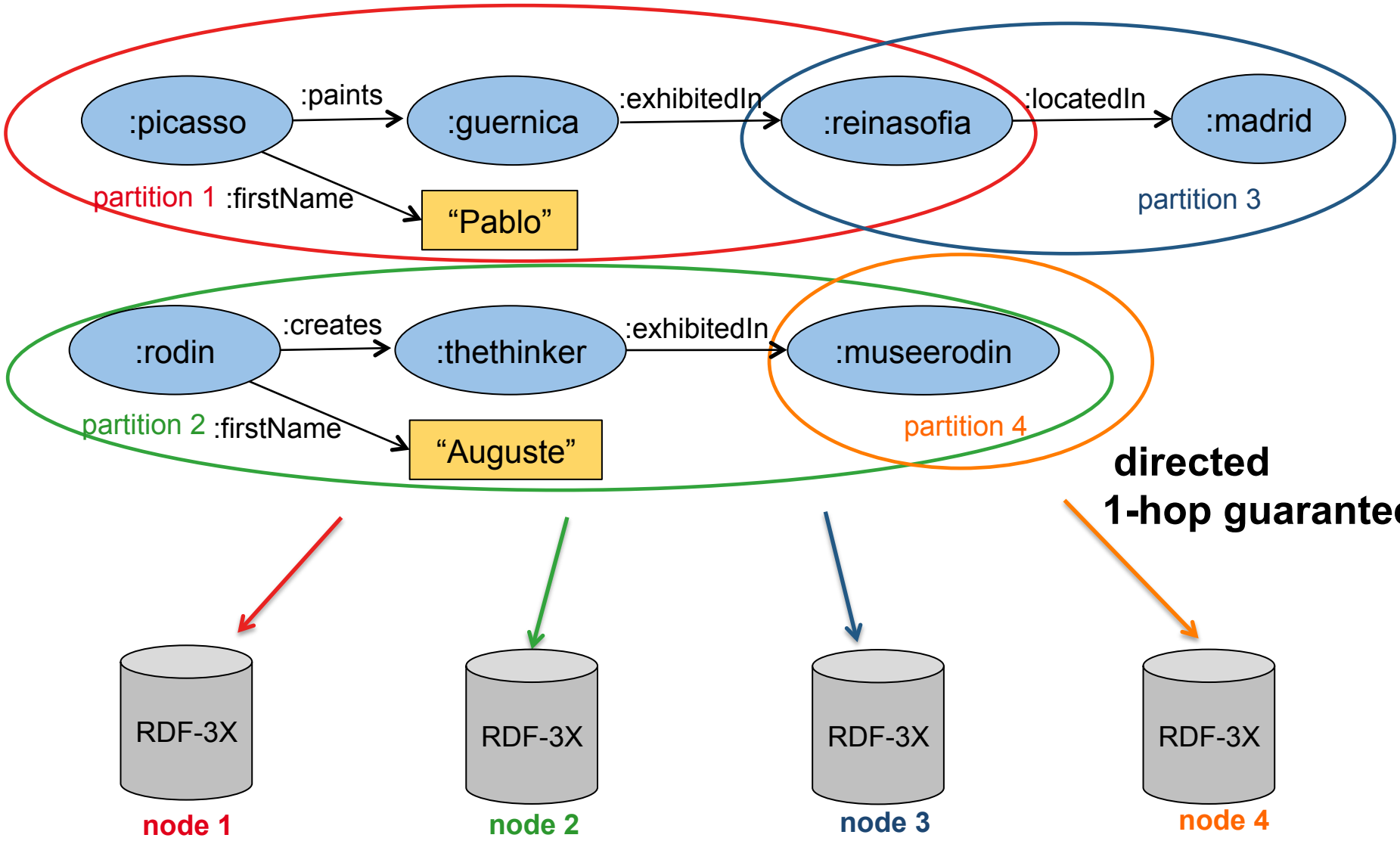
- Data partitioning
  - METIS graph partitioner [METIS]
  - vertex partitioning
  - #partitions = #machines
  - minimum edge cut
  - **`type`** triples are excluded from the partitioning process
- Data placement
  - place the triple in the partition in which the subject belongs to
  - replication across the partitions
  - directed/undirected n-hop guarantee
- Data storage
  - each machine stores triples into RDF-3X



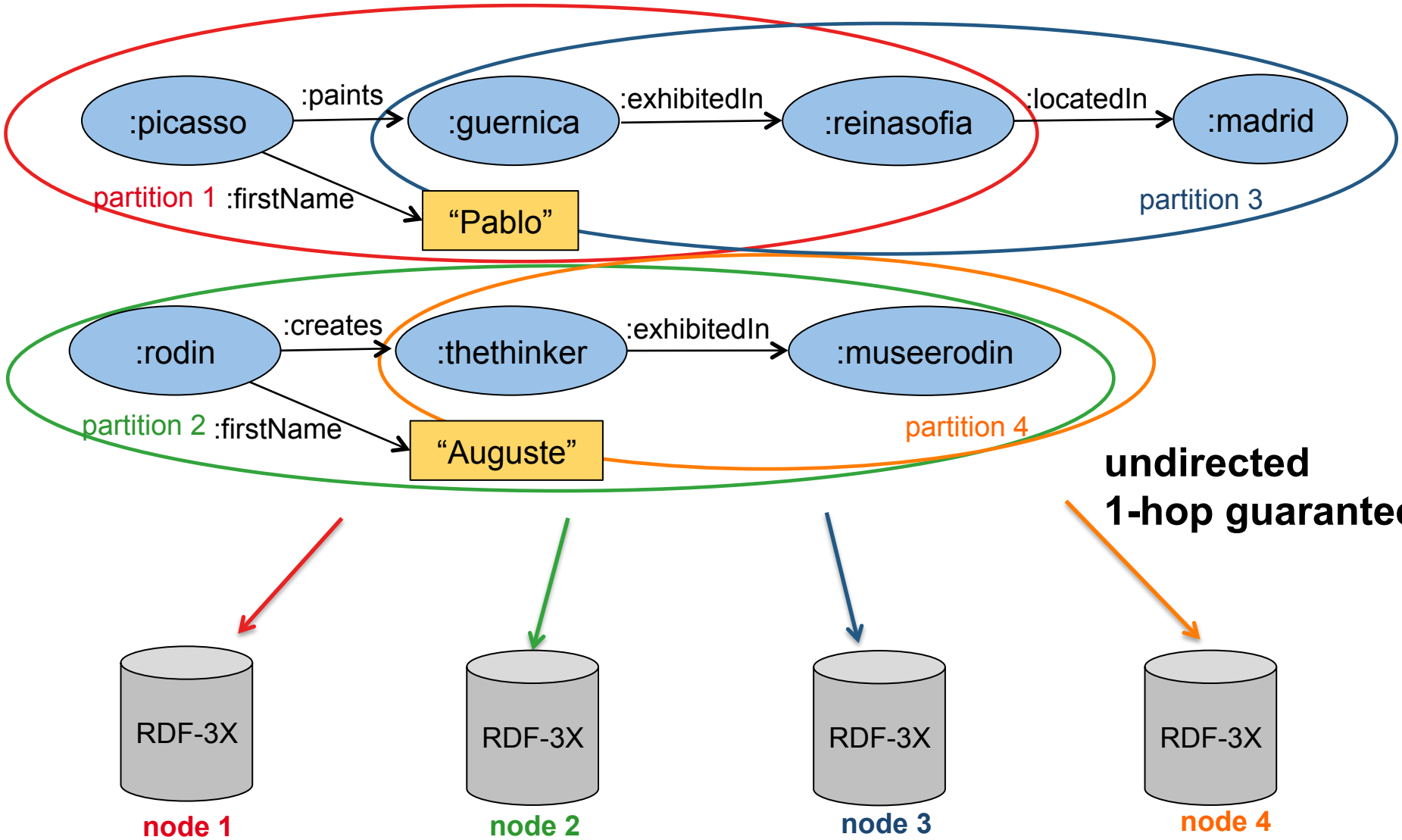
Graph partitioning [Huang11] - storage



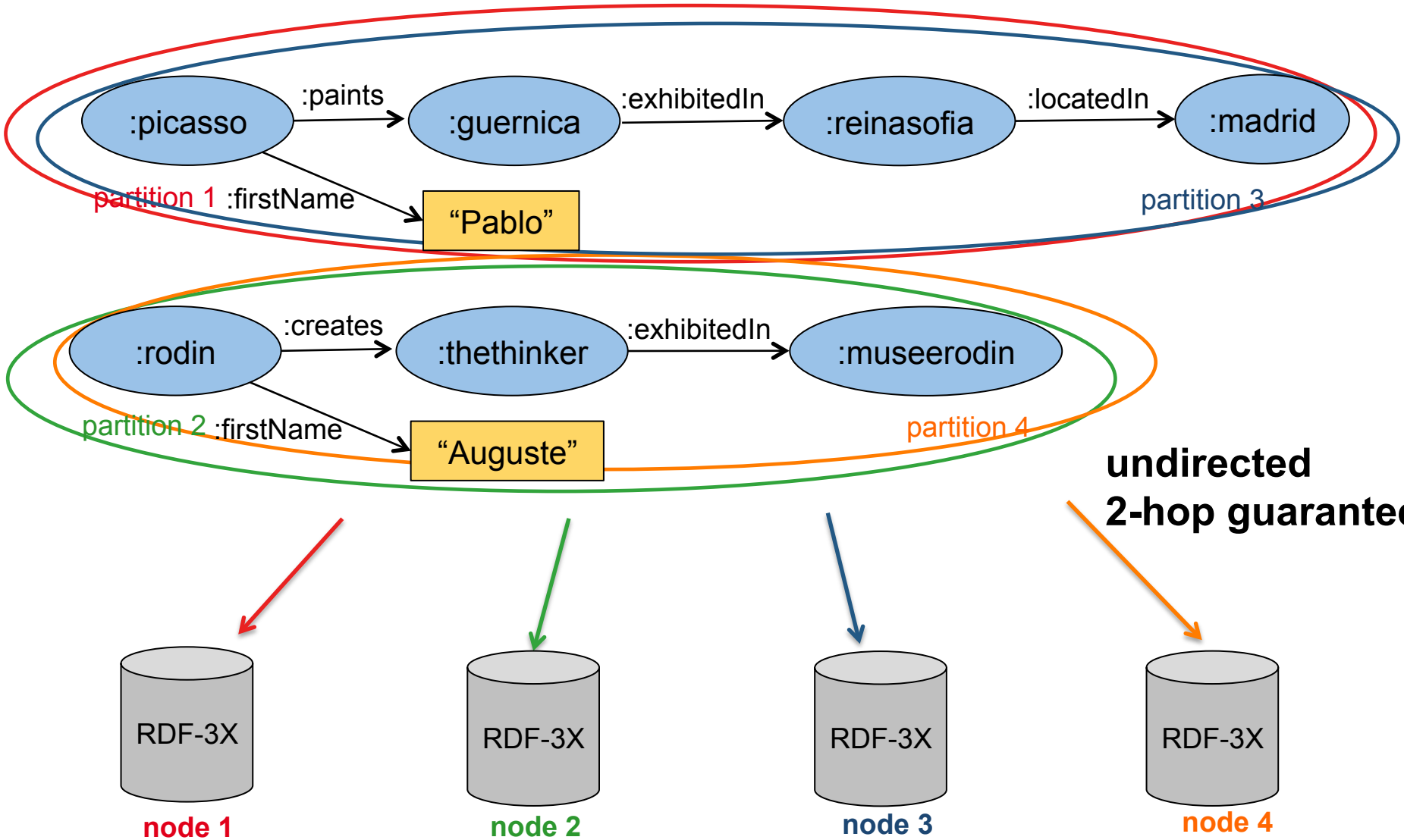
Graph partitioning [Huang11] - storage



Graph partitioning [Huang11] - storage

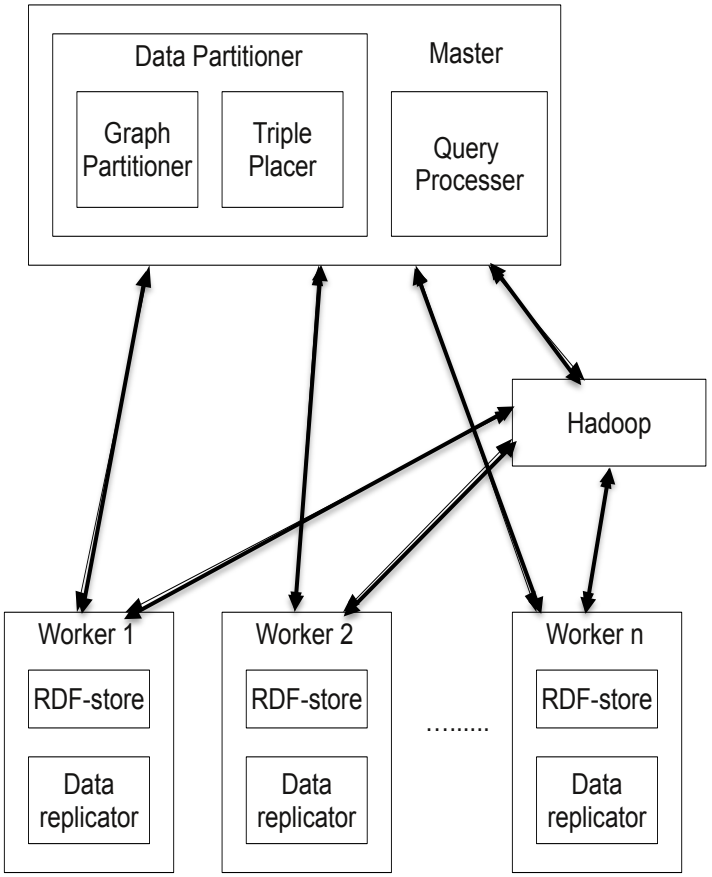


Graph partitioning [Huang11] - storage



Graph partitioning [Huang11] - querying

- Query decomposition for parallelization
- Check if q is PWOC (parallelizable without communication)
  - Yes: answer only from RDF-3X and union results
  - No: answer subqueries from RDF-3X and join them with Hadoop
- # jobs = # subqueries
- Heuristic optimization
  - decomposition so as to have minimum number of subqueries → minimal edge partitioning of a graph into subgraphs of bounded diameter

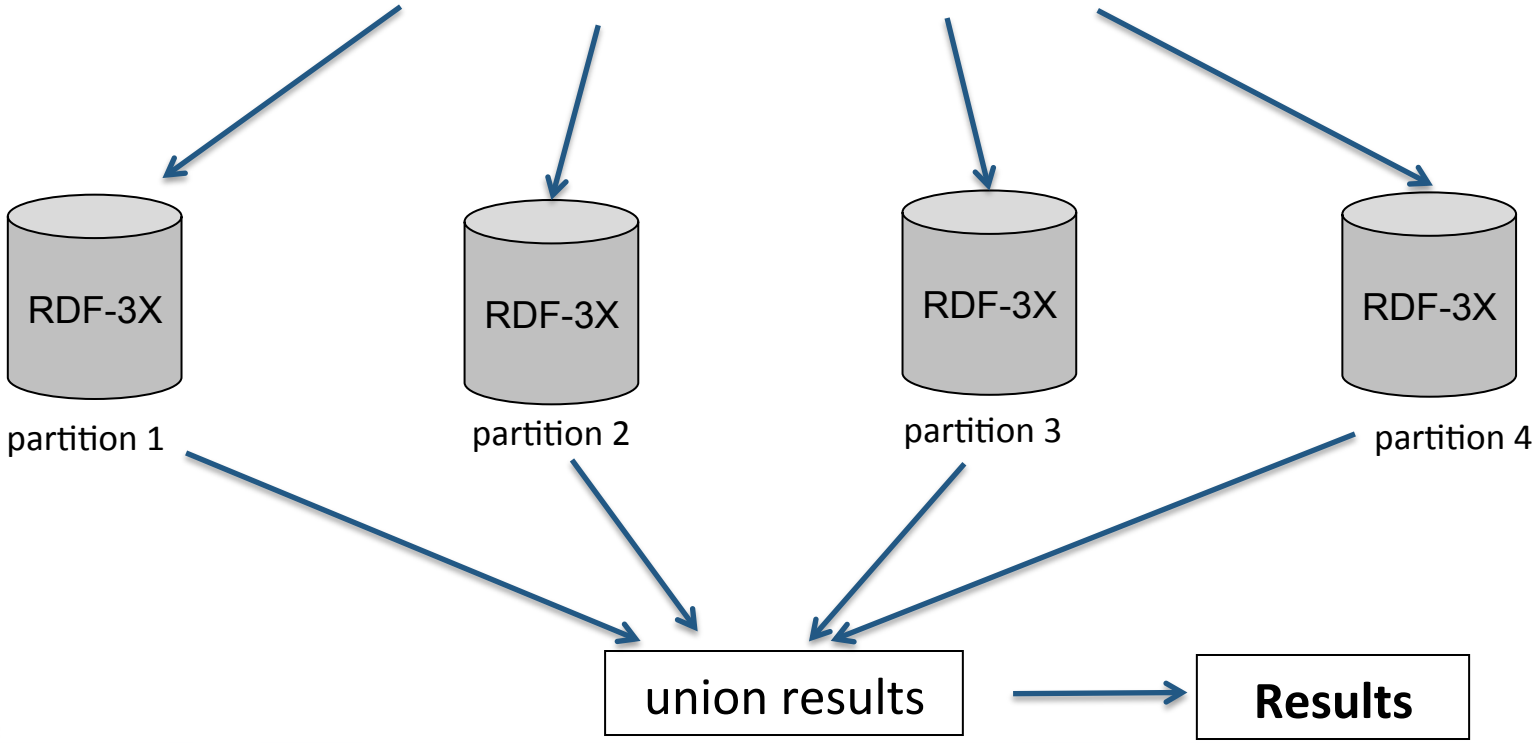


Graph partitioning [Huang11] - querying

replication with
1-hop guarantee

PWOC query

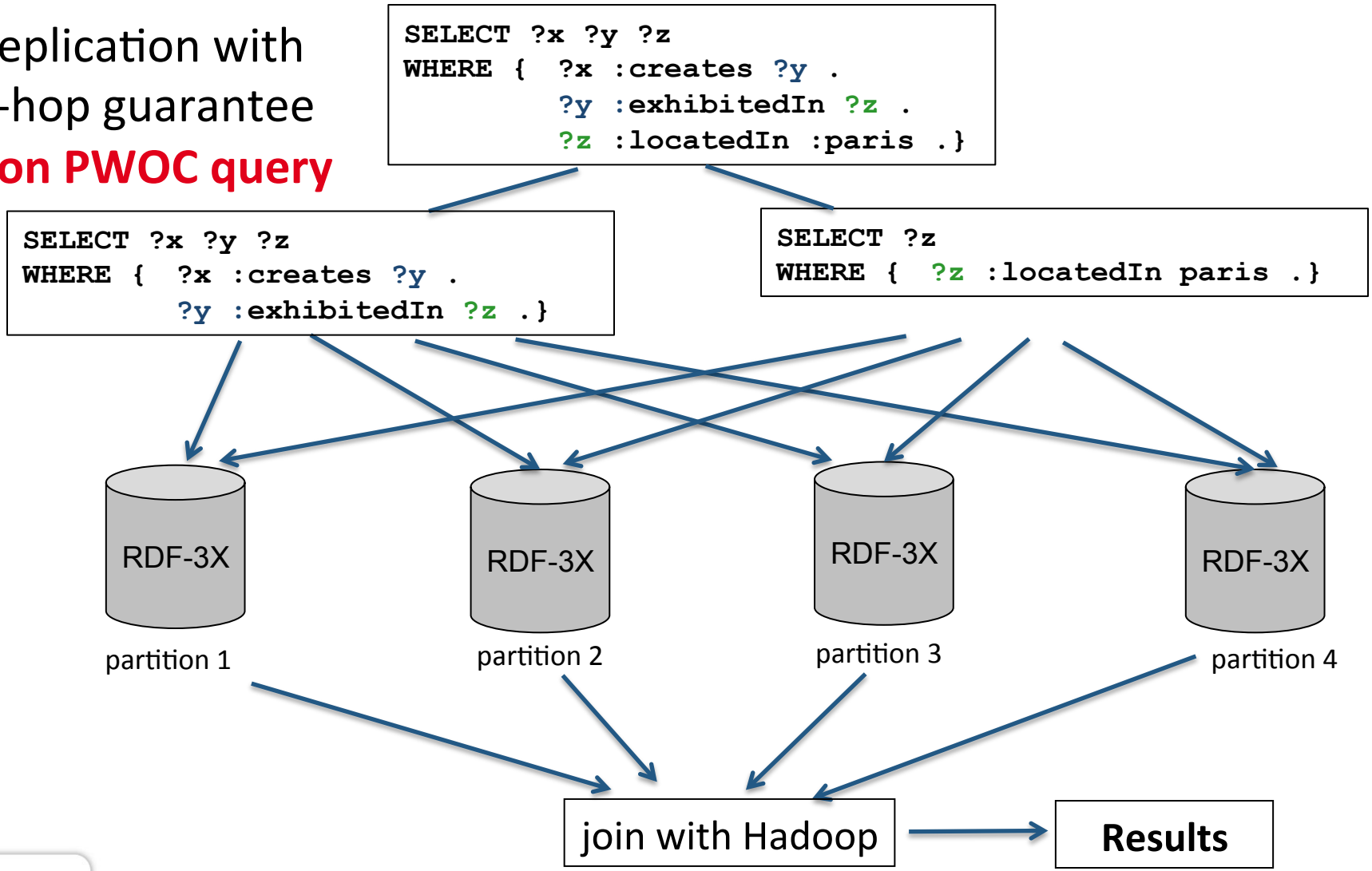
```
SELECT ?x ?y ?z
WHERE {
        ?x type :artist .
        ?x :firstName ?y .
        ?x :creates ?z .}
```



Graph partitioning [Huang11] - querying

replication with
1-hop guarantee

non PWOC query

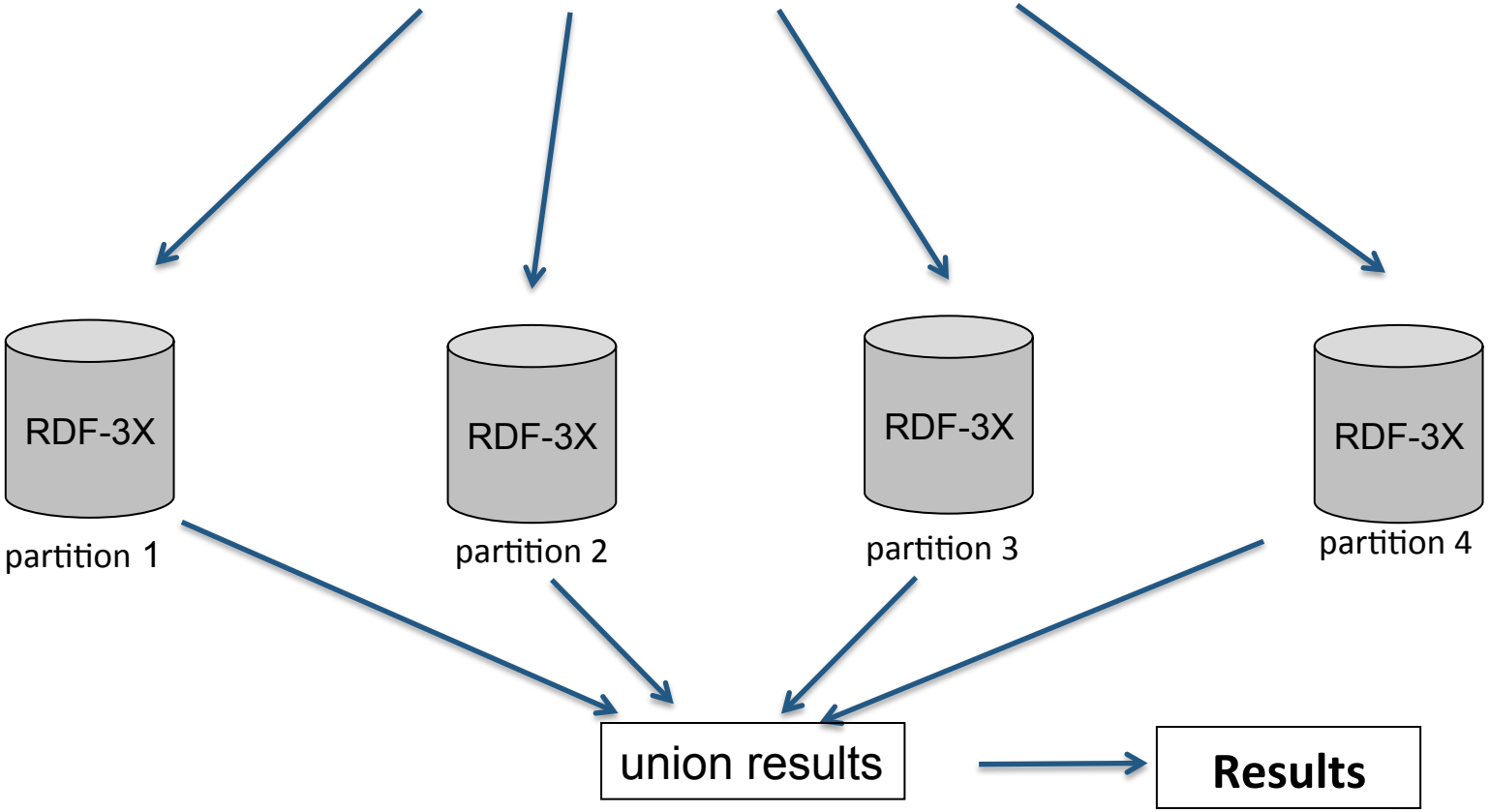


Graph partitioning [Huang11] - querying

replication with
2-hop guarantee

PWOC query

```
SELECT ?x ?y ?z
WHERE {  ?x :creates ?y .
         ?y :exhibitedIn ?z .
         ?z :locatedIn :paris .}
```



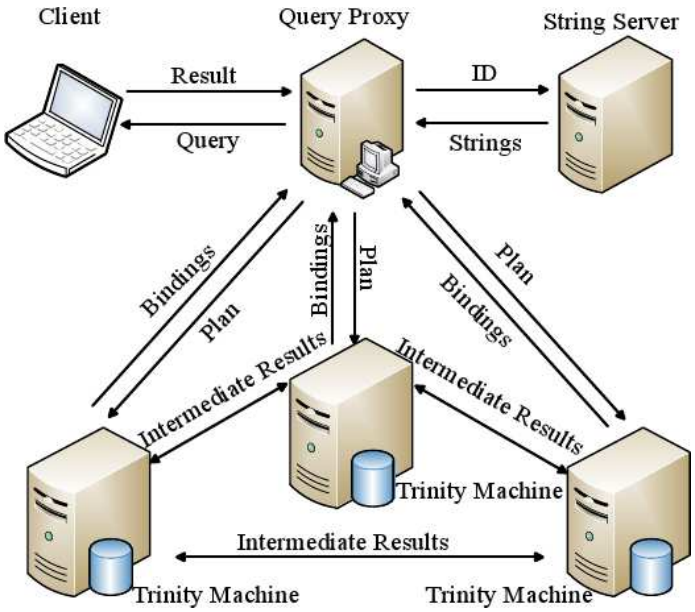
Graph partitioning [Huang11] - conclusions

- ✚ Benefits from RDF structure
- ✚ Data placement
- ✚ Reduces shuffling

- ━ Not scalable for big datasets
- ━ Subqueries go to all machines
- ━ Uses a lot of resources

Trinity.RDF [Zeng13]

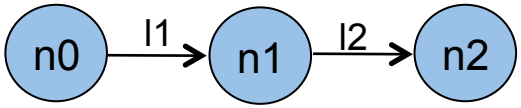
- Built on top of a memory cloud (in-memory key-value store)
- Uses Trinity: a memory-based distributed graph platform
- Takes advantage of the graph-based model of RDF for query processing
- Graph exploration instead of joins (pruning of candidate matches in a greedy manner)
  - only if graph exploration can be implemented more efficiently than join
  - similar to index-nested-loops join



How can we model a graph in a key-value store?

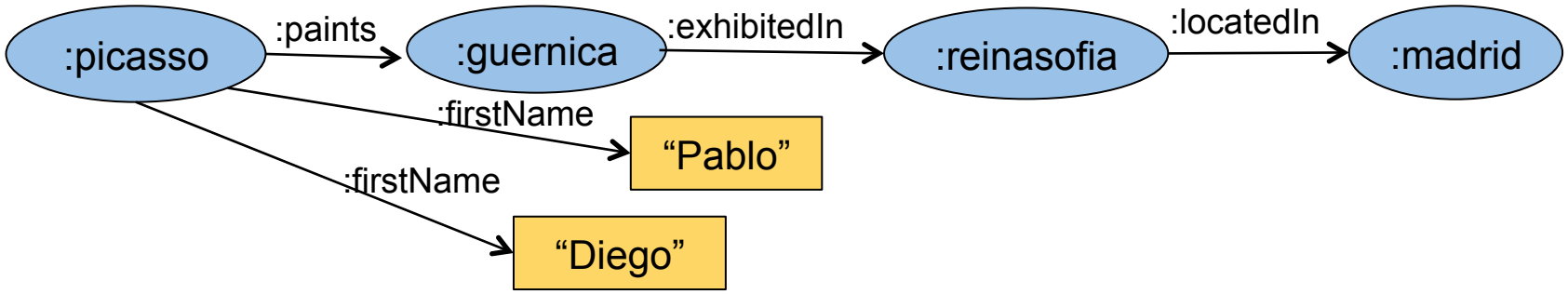
- Each node is assigned to a machine through hashing
- Two modeling choices
  - 1. (key, value) : (node_id, <in-adjacency-list, out-adjacency-list>)

(property, subject) (property, object)

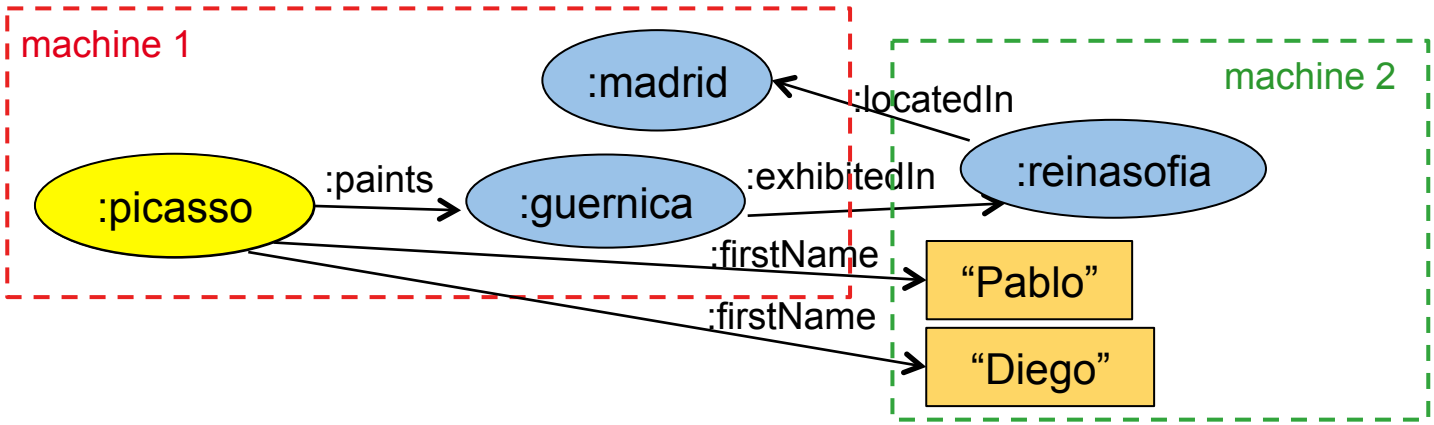


(n1, <{(l1, n0)} {(l2,n2)}>)
(n0, <{-} {(l1,n1)}>)
(n2, <{(l2,n2)} {-}>)

Modeling graphs in a key-value store (model 1)



Modeling graphs in a key-value store (model 1)



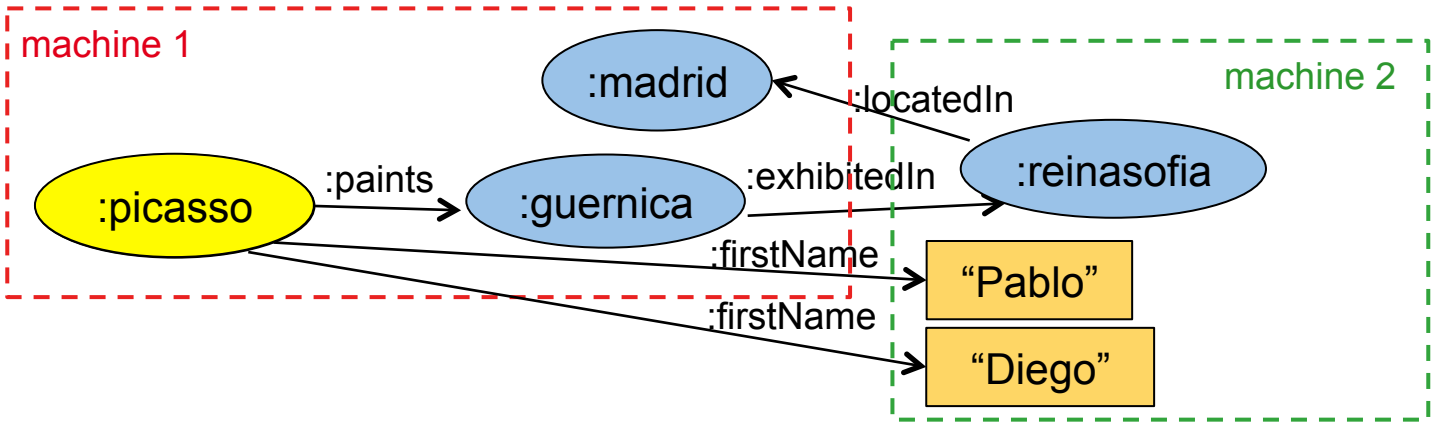
machine 1

Key	Value
:picasso	In: - Out: (:paints, :guernica) **Out: (:firstName, "Pablo")** **Out: (:firstName, "Diego")**
:guernica	In: (:paints, :picasso) **Out: (:exhibitedIn, :reinasofia)**
:madrid	**Out: (:locatedIn, :reinasofia)**

machine 2

Key	Value
:reinasofia	**In: (:exhibitedIn, :guernica)** **Out: (:locatedIn, :madrid)**
"Pablo"	**In: (:firstName, :picasso)**
"Diego"	**In: (:firstName, :picasso)**

Modeling graphs in a key-value store (model 1)



machine 1

Key	Value
:picasso	In: - Out: (:paints, :guernica) **Out: (:firstName, “Pablo”)** **Out: (:firstName, “Diego”)**
:guernica	In: (:paints, :picasso) **Out: (:exhibitedIn, :reinasofia)**
:madrid	**Out: (:locatedIn, :reinasofia)**

machine 2

Key	Value
:reinasofia	**In: (:exhibitedIn, :guernica)** **Out: (:locatedIn, :madrid)**
“Pablo”	**In: (:firstName, :picasso)**
“Diego”	**In: (:firstName, :picasso)**

2 messages should be sent to continue the node exploration from *:picasso* through *:firstName*

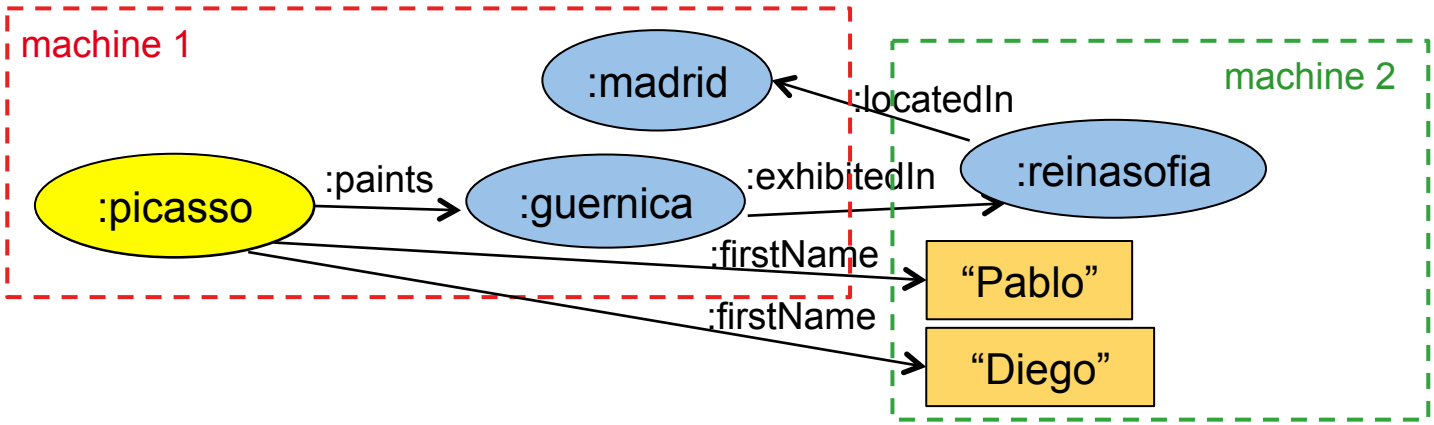
How can we model a graph in a key-value store?

- Each node is assigned to a machine through hashing
- Two modeling choices
  - 1. (key, value) : (node_id, <in-adjacency-list, out-adjacency-list>)
  - 2. (key, value): (node_id, <in1, in2…., out1, out2…>)

    (in_i, in-adjacency-list$_i$): on the same machine i

    (out_i, out-adjacency-list$_i$): on the same machine I

Modeling graphs in a key-value store (model 2)



machine 1

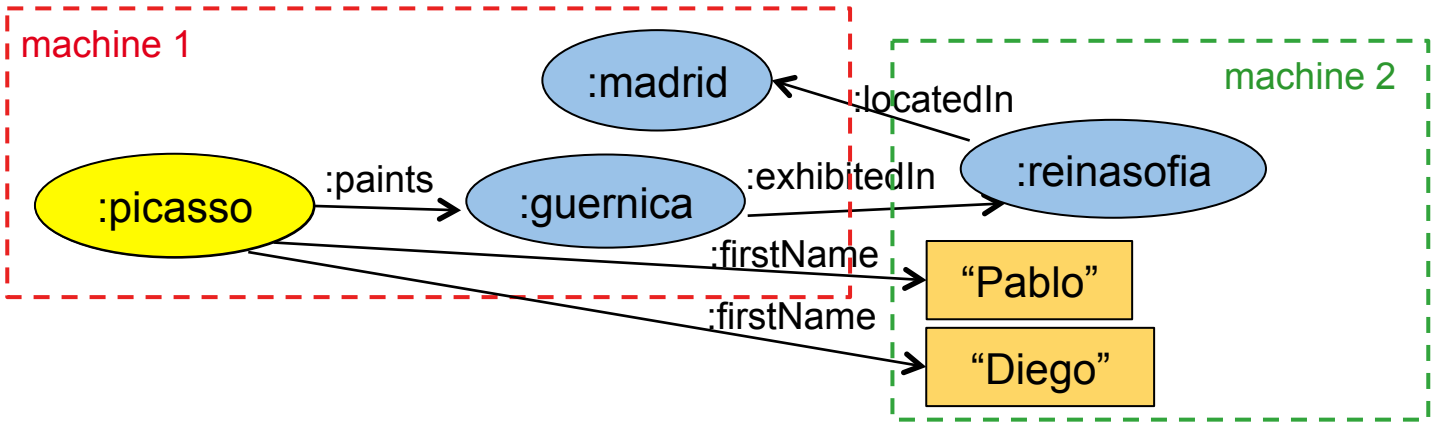
Key	Value
:picasso	out1 out2
:guernica	in1 out3
:madrid	out4
out1	(:paints, :guernica)
in1	(:paints, :picasso)
in2	(:exhibitedIn, :guernica)
out5	(:locatedIn, :madrid)
in3	(:firstName, :picasso)

machine 2

Key	Value
:reinasofia	in2 out5
"Pablo"	in3
out2	**(:firstName, "Pablo)** **(:firstName, "Diego")**
out3	(:exhibitedIn, :reinasofia)
out4	(:locatedIn, :reinasofia)

only 1 message needs to be sent

Modeling graphs in a key-value store (model 2)



machine 1

Key	Value
:picasso	ou ou
:guernica	in ou
:madrid	ou
out1	(:paints, :guernica)
in1	(:paints, :picasso)
in2	(:exhibitedIn, :guernica)
out5	(:locatedIn, :madrid)
in3	(:firstName, :picasso)

machine 2

Key	Value
	2 5
	3
	ne, "Pablo) (:firstName, "Diego")
out3	(:exhibitedIn, :reinasofia)
out4	(:locatedIn, :reinasofia)

No more than *#machines* messages are sent

only 1 message needs to be sent

How can we model an RDF graph in a key-value store?

- Each node is assigned to a machine through hashing
- Two modeling choices
  - 1. (key, value) : (node_id, <in-adjacency-list, out-adjacency-list>)
  - 2. (key, value): (node_id, <in1, in2…., out1, out2…>)

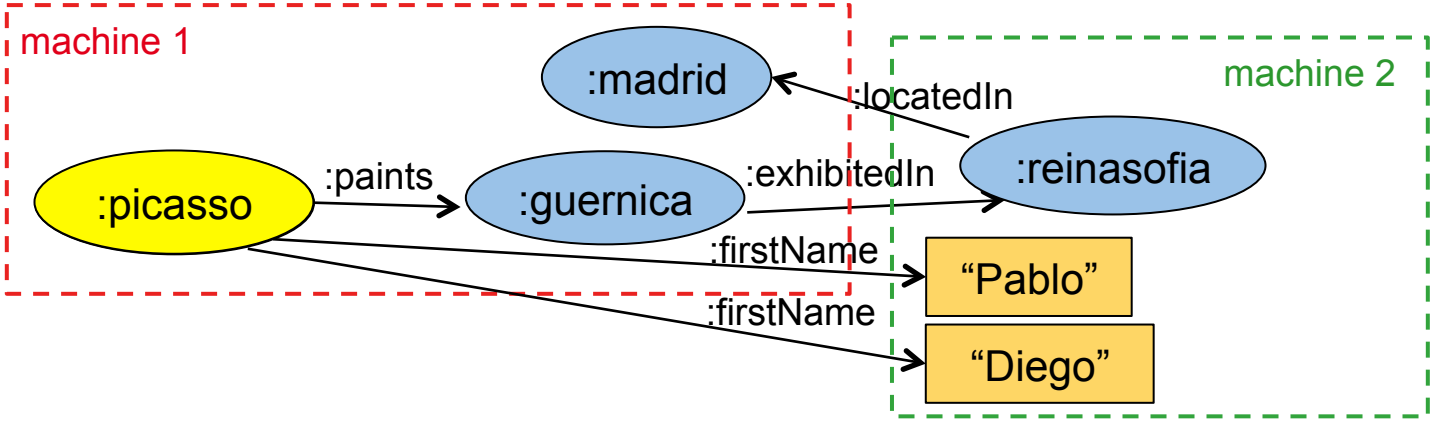
    (in_i, in-adjacency-list$_i$): on the same machine i

    (out_i, out-adjacency-list$_i$): on the same machine I
- Threshold on the number of neighbors of a node to decide whether to use model (1) or (2)
- Local index on the predicate
- Like SPO and OPS indexes

How can we model an RDF graph in a key-value store?

- *What happens for queries of the form (?x, p, ?y)*
- Global predicate indexing (predicate, ⟨subject-list$_i$, object-list$_i$⟩)

unique subjects on machine i unique objects on machine i



machine 1

Key	Value
:paints	{(:picasso)} {(:guernica)}
:firstName	{(:picasso)} {-}
:exhibitedIn	{(:guernica)} {-}
:locatedIn	{-} {(:madrid)}

machine 2

Key	Value
:firstName	{-} {("Pablo", "Diego")}
:exhibitedIn	{-} {(:reinasofia)}
:locatedIn	{(:reinasofia)} {-}

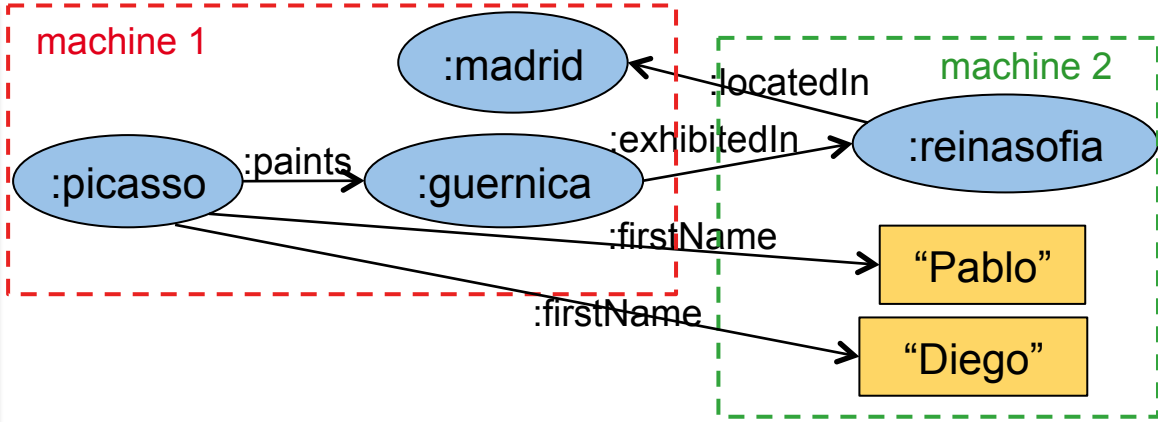
Accessing RDF graphs

- Single triple pattern (s, p, o)
  - s or o is constant → start exploring the graph from this node
  - s and o is variable →
    - Use POS to find the nodes that are connected by the edge p
    - For each node → graph exploration and filtering out edges that aren't p

Matching triple patterns

Key	Value
:picasso	In: - Out: (:paints, :guernica) **Out: (:firstName, "Pablo")** **Out: (:firstName, "Diego")**
:guernica	In: (:paints, :picasso) **Out: (:exhibitedIn, :reinasofia)**
:madrid	**Out: (:locatedIn, :reinasofia)**
:paints	{(:picasso)} {(:guernica)}
:firstName	{(:picasso)} {-}
:exhibitedIn	{(:guernica)} {-}
:locatedIn	{-} {(:madrid)}

Key	Value
:reinasofia	**In: (:exhibitedIn, :guernica)** **Out: (:locatedIn, :madrid)**
"Pablo"	**In: (:firstName, :picasso)**
"Diego"	**In: (:firstName, :picasso)**
:firstName	{-} {("Pablo", "Diego")}
:exhibitedIn	{-} {(:reinasofia)}
:locatedIn	{(:reinasofia)} {-}

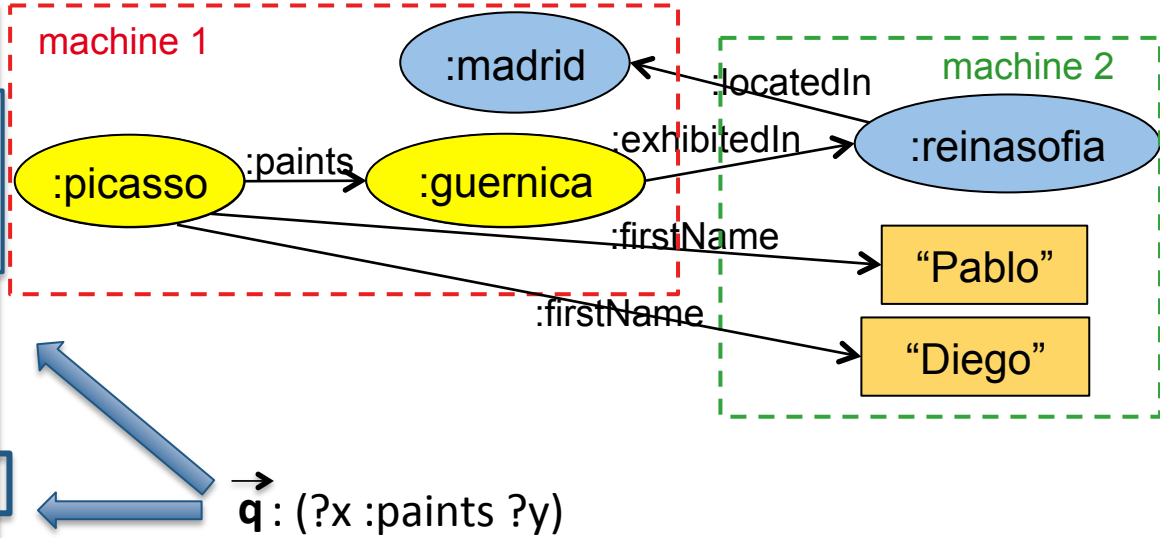


$\vec{q}$: (?x :paints ?y)

Matching triple patterns

Key	Value
:picasso	In: - Out: (:paints, :guernica) **Out: (:firstName, "Pablo")** **Out: (:firstName, "Diego")**
:guernica	In: (:paints, :picasso) **Out: (:exhibitedIn, :reinasofia)**
:madrid	**Out: (:locatedIn, :reinasofia)**
:paints	{(:picasso)} {(:guernica)}
:firstName	{(:picasso)} {-}
:exhibitedIn	{(:guernica)} {-}
:locatedIn	{-} {(:madrid)}

Key	Value
:reinasofia	**In: (:exhibitedIn, :guernica)** **Out: (:locatedIn, :madrid)**
"Pablo"	**In: (:firstName, :picasso)**
"Diego"	**In: (:firstName, :picasso)**
:firstName	{-} {("Pablo", "Diego")}
:exhibitedIn	{-} {(:reinasofia)}
:locatedIn	{(:reinasofia)} {-}



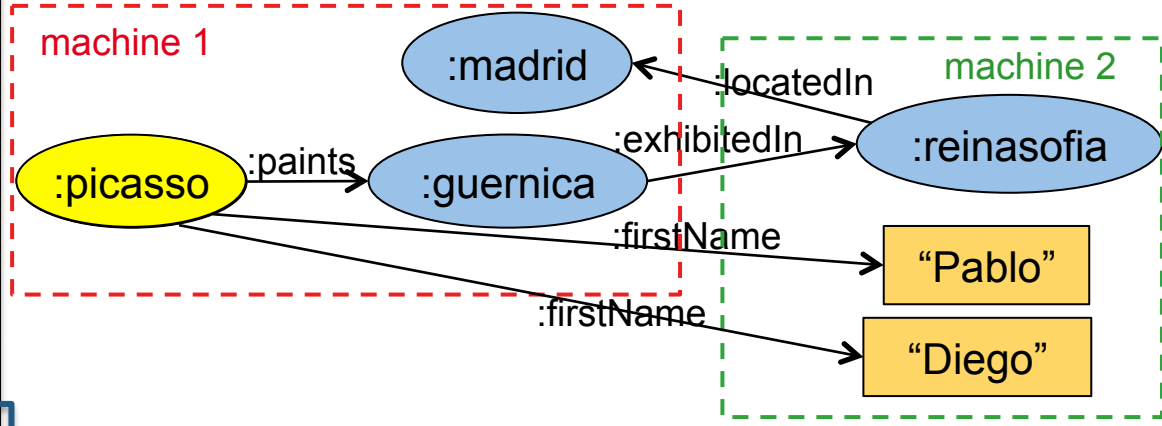
Accessing RDF graphs

- Single triple pattern (s, p, o)
  - s or o is constant → start exploring the graph from this node
  - s and o is variable →
    - Use POS to find the nodes that are connected by the edge p
    - For each node → graph exploration and filtering out edges that aren't p
- Multiple triple patterns:
  - process sequentially by graph exploration
  - prunes intermediate results

Joins by exploring the graph

Key	Value
:picasso	In: - Out: (:paints, :guernica) **Out: (:firstName, "Pablo")** **Out: (:firstName, "Diego")**
:guernica	In: (:paints, :picasso) **Out: (:exhibitedIn, :reinasofia)**
:madrid	**Out: (:locatedIn, :reinasofia)**
:paints	{(:picasso)} {(:guernica)}
:firstName	{(:picasso)} {-}
:exhibitedIn	{(:guernica)} {-}
:locatedIn	{-} {(:madrid)}

Key	Value
:reinasofia	**In: (:exhibitedIn, :guernica)** **Out: (:locatedIn, :madrid)**
"Pablo"	**In: (:firstName, :picasso)**
"Diego"	**In: (:firstName, :picasso)**
:firstName	{-} {("Pablo", "Diego")}
:exhibitedIn	{-} {(:reinasofia)}
:locatedIn	{(:reinasofia)} {-}

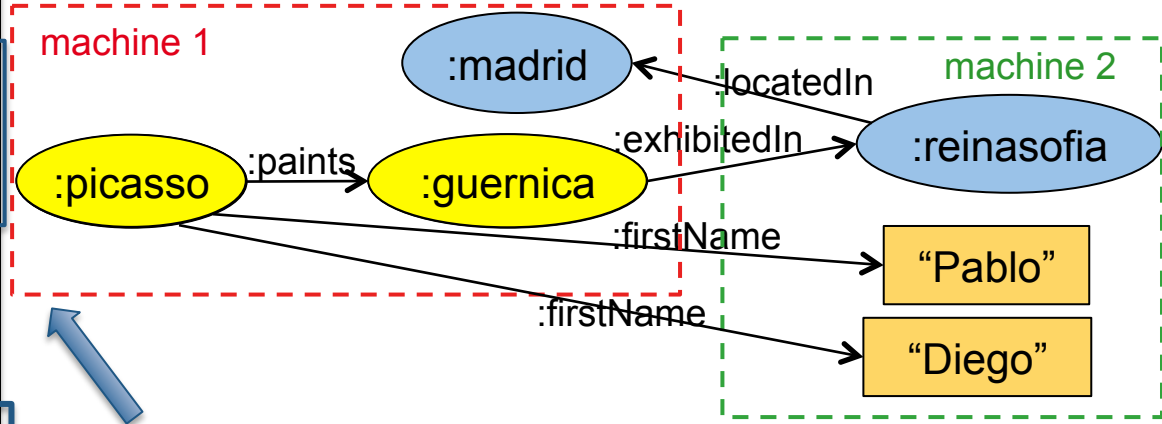


$\vec{q}$: (?x :paints ?y) ∧
(?y, exhibitedIn, ?z) ∧
(?z locatedIn, :madrid)

Joins by exploring the graph

Key	Value
:picasso	In: - Out: (:paints, :guernica) **Out: (:firstName, "Pablo")** **Out: (:firstName, "Diego")**
:guernica	In: (:paints, :picasso) **Out: (:exhibitedIn, :reinasofia)**
:madrid	**Out: (:locatedIn, :reinasofia)**
:paints	{(:picasso)} {(:guernica)}
:firstName	{(:picasso)} {-}
:exhibitedIn	{(:guernica)} {-}
:locatedIn	{-} {(:madrid)}

Key	Value
:reinasofia	**In: (:exhibitedIn, :guernica)** **Out: (:locatedIn, :madrid)**
"Pablo"	**In: (:firstName, :picasso)**
"Diego"	**In: (:firstName, :picasso)**
:firstName	{-} {("Pablo", "Diego")}
:exhibitedIn	{-} {(:reinasofia)}
:locatedIn	{(:reinasofia)} {-}

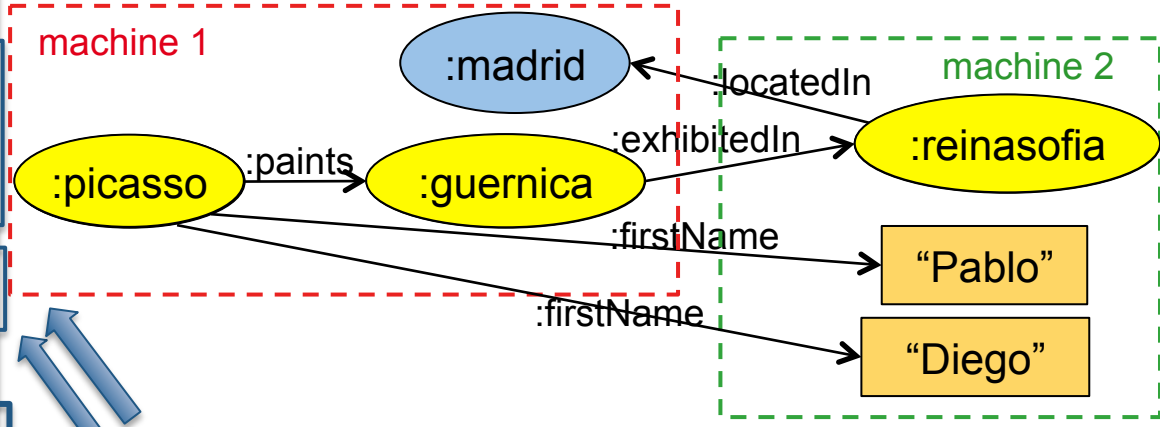


$\vec{q}$: (?x :paints ?y) ∧
(?y, exhibitedIn, ?z) ∧
(?z locatedIn, :madrid)

Joins by exploring the graph

Key	Value
:picasso	In: - Out: (:paints, :guernica) **Out: (:firstName, "Pablo")** **Out: (:firstName, "Diego")**
:guernica	In: (:paints, :picasso) **Out: (:exhibitedIn, :reinasofia)**
:madrid	**Out: (:locatedIn, :reinasofia)**
:paints	{(:picasso)} {(:guernica)}
:firstName	{(:picasso)} {-}
:exhibitedIn	{(:guernica)} {-}
:locatedIn	{-} {(:madrid)}

Key	Value
:reinasofia	**In: (:exhibitedIn, :guernica)** **Out: (:locatedIn, :madrid)**
"Pablo"	**In: (:firstName, :picasso)**
"Diego"	**In: (:firstName, :picasso)**
:firstName	{-} {("Pablo", "Diego")}
:exhibitedIn	{-} {(:reinasofia)}
:locatedIn	{(:reinasofia)} {-}

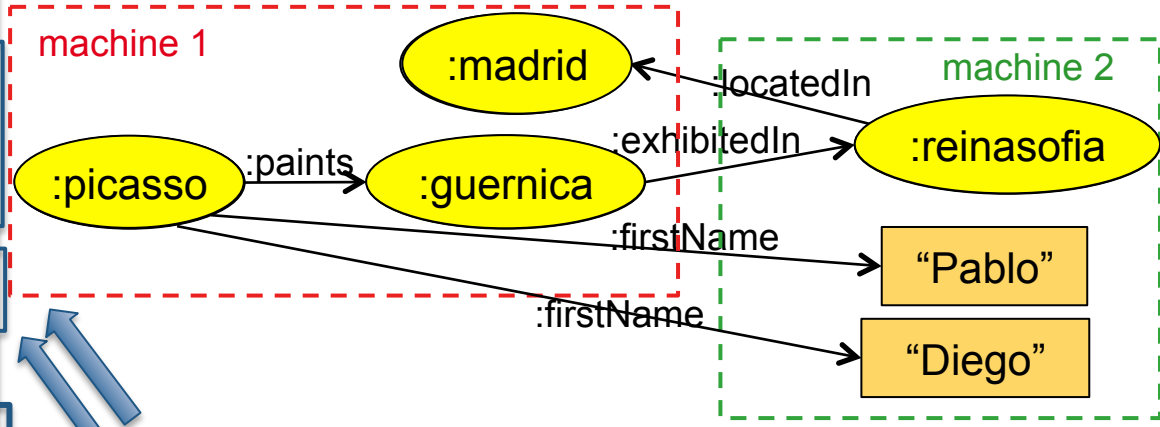


$\vec{q}$: (?x :paints ?y) ∧
(?y, exhibitedIn, ?z) ∧
(?z locatedIn, :madrid)

Joins by exploring the graph

Key	Value
:picasso	In: - Out: (:paints, :guernica) **Out: (:firstName, "Pablo")** **Out: (:firstName, "Diego")**
:guernica	In: (:paints, :picasso) **Out: (:exhibitedIn, :reinasofia)**
:madrid	**Out: (:locatedIn, :reinasofia)**
:paints	{(:picasso)} {(:guernica)}
:firstName	{(:picasso)} {-}
:exhibitedIn	{(:guernica)} {-}
:locatedIn	{-} {(:madrid)}

Key	Value
:reinasofia	**In: (:exhibitedIn, :guernica)** **Out: (:locatedIn, :madrid)**
"Pablo"	**In: (:firstName, :picasso)**
"Diego"	**In: (:firstName, :picasso)**
:firstName	{-} {("Pablo", "Diego")}
:exhibitedIn	{-} {(:reinasofia)}
:locatedIn	{(:reinasofia)} {-}



$\vec{q}$: (?x :paints ?y) ∧
(?y, exhibitedIn, ?z) ∧
(?z locatedIn, :madrid)

Accessing RDF graphs

- Single triple pattern (s, p, o)
  - s or o is constant → start exploring the graph from this node
  - s and o is variable →
    - Use POS to find the nodes that are connected by the edge p
    - For each node → graph exploration and filtering out edges that aren't p
- Multiple triple patterns:
  - process sequentially by graph exploration
  - prunes intermediate results
- Final join in a proxy for removing invalid results (small join)
- Ordering matters!

Trinity.RDF - results

	L1	L2	L3	L4	L5	L6	L7	Geo. mean
Trinity.RDF	**12648**	6018	8735	**5**	**4**	**9**	31214	**450**
RDF-3X (Warm Cache)	36m47s	14194	27245	8	8	65	69560	2197
BitMat (Warm Cache)	33097	209146	**2538**	aborted	407	1057	aborted	5966
RDF-3X (Cold Cache)	39m2s	18158	34241	1177	1017	993	98846	15003
BitMat (Cold Cache)	39716	225640	9114	aborted	494	2151	aborted	9721
MapReduce-RDF-3X (Warm Cache)	17188	**3164**	16932	14	10	720	**8868**	973
MapReduce-RDF-3X (Cold Cache)	32511	7371	19328	675	770	1834	19968	5087

Table 11: Query run-times in milliseconds for the LUBM-10240 dataset (1.36 billion triples)

	L1	L2	L3	L4	L5	L6	L7	Geo. mean
Trinity.RDF	176	21	119	**0.005**	**0.006**	**0.010**	126	**1.494**
RDF-3X (Warm Cache)	aborted	96	363	0.011	0.006	0.021	548	1.726
RDF-3X (Cold Cache)	aborted	186	1005	874	578	981	700	633.842
MapReduce-RDF-3X (Warm Cache)	**102**	**19**	**113**	0.022	0.016	0.226	**51.98**	2.645
MapReduce-RDF-3X (Cold Cache)	171	32	151	1.113	0.749	1.428	89	13.633

Table 12: Query run-times in seconds for the LUBM-100000 dataset (9.96 billion triples)

5-nodes and 8-nodes cluster with each machine:
- 96 GB RAM,
- two 2.67 GHz CPUs (6 cores and 12 threads each)

40Gb/s InfiniBand Network adaptor

IV. RDF stores based on cloud services

IV. RDF stores based on cloud services

- Statustore [Stein10]
  - AWS SimpleDB
- AMADA [Bugiotti12, Aranda12]
  - Generic cloud-based architecture
  - Built on top of Amazon Web Services as a SaaS
  - Route the queries to a small subset of the data that are likely to have matches
  - Trade-offs between efficiency and monetary costs

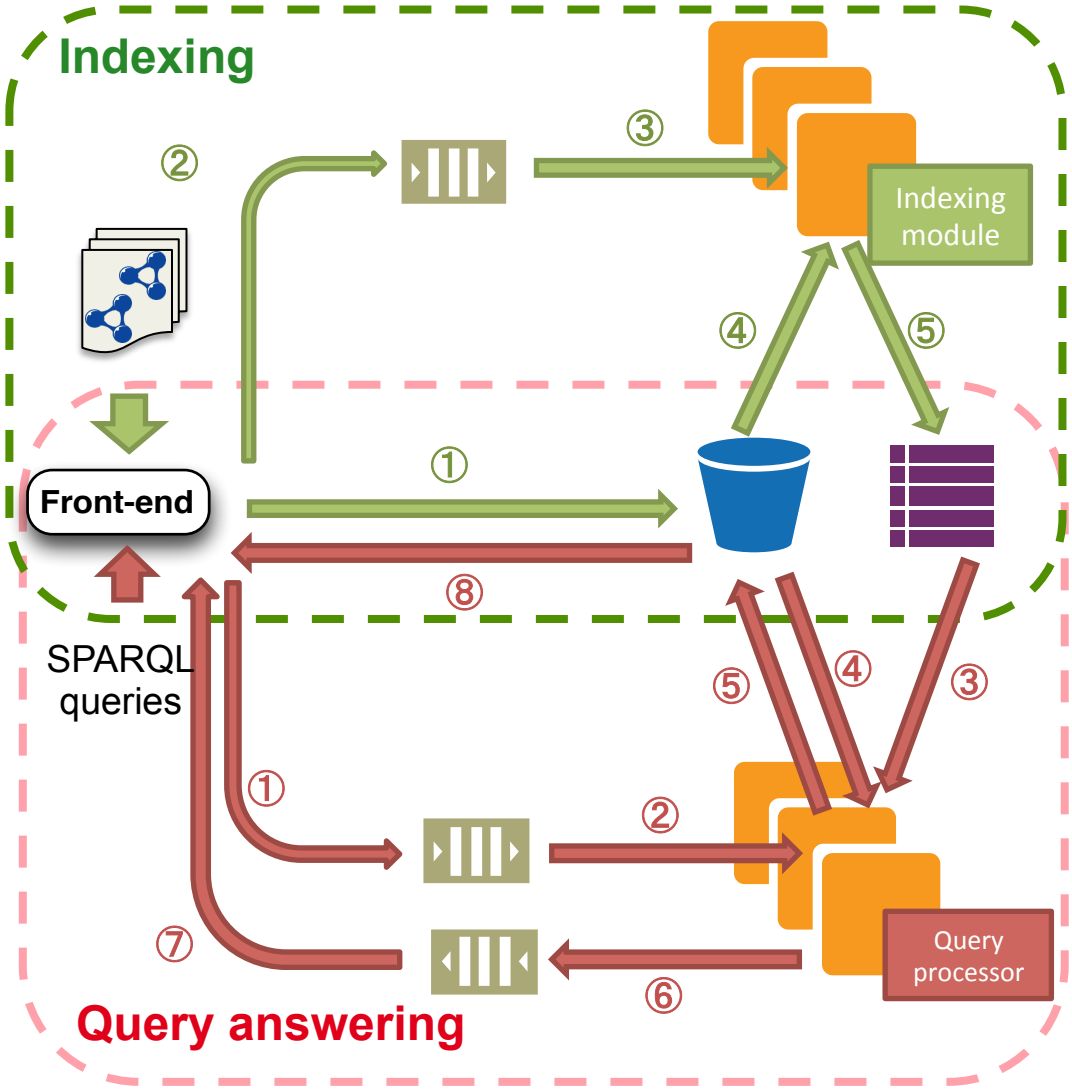
AMADA architecture

Key-value store
Ideal for indexing and querying small, structured data

Storage for raw data
Ideal for large files

Virtual machines
Resizable computing capacity in the cloud

Messaging queues
Queues for communication between distributed components



AMADA indexing and storage

S table

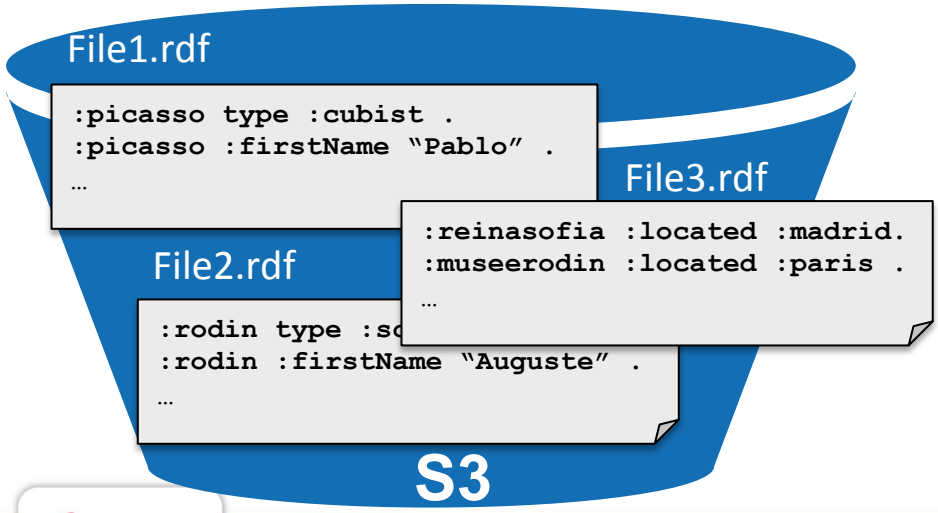
key	(attribute, value)
:picasso	(file1, -)
:guernica	(file1, -)
:reinasofia	(file1, -)
:rodin	(file2, -)
:thethinker	(file2, -)
:museerodin	(file2, -)
…	…

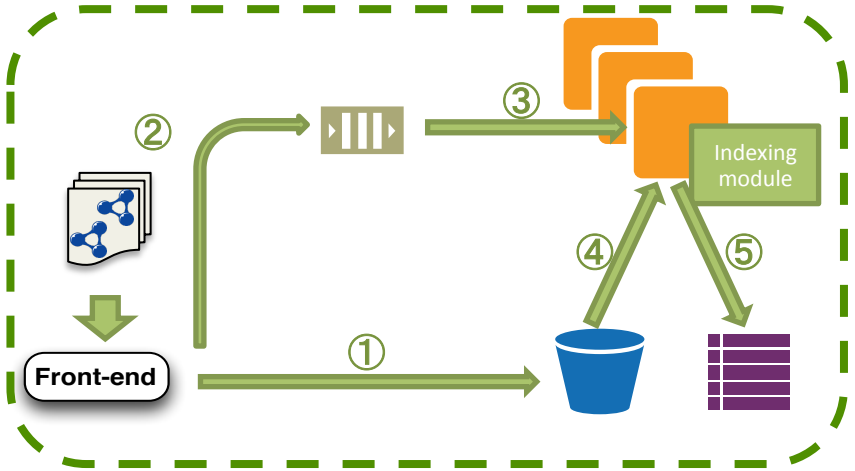
P table

key	(attribute, value)
type	(file1,-), (file2, -)
:firstName	(file1,-), (file2, -)
:paints	(file1,-)
:exhibitedIn	(file1,-), (file2, -)
:locatedIn	(file1,-)
:creates	(file2,-)
…	…

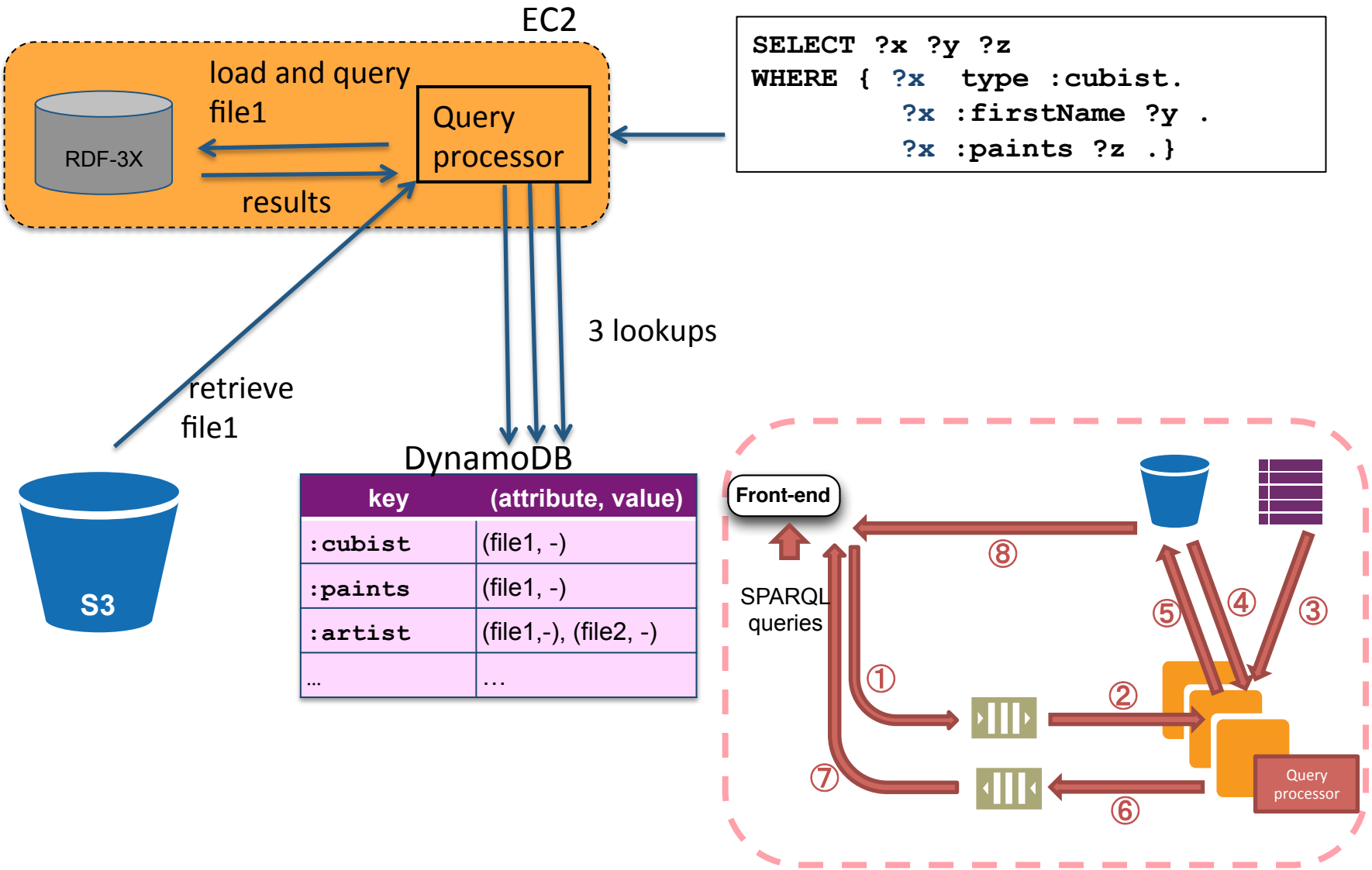
O table

key	(attribute, value)
:cubist	(file1,-)
:guernica	(file1,-)
"Pablo"	(file1,-)
:sculptor	(file2,-)
:thethinker	(file2,-)
:museerodin	(file2,-)
…	…





AMADA querying



AMADA indexing strategies

- Source indexing strategies
  - Identify files (F) that contribute an answer to a query
  - Term-based: T|F|- (T: URIs, literals etc. without caring if it's S, P, or O)
  - Attribute-based: S|F|- P|F|- O|F|-
  - Attribute-subset: S|F|- P|F|- O|F|- SP|F|- PS|F|- SO|F|- SPO|F|-
- Distributed data indexing (as in Part II)
  - RDF data itself is indexed in the key-value store
  - S|P|O P|O|S O|S|P
- Different trade-offs

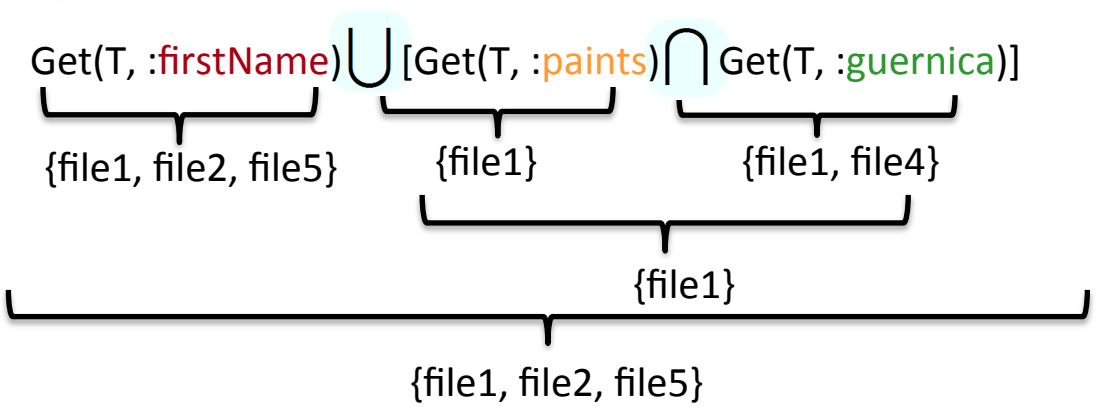
Term-based example

T table

key	(attribute, value)
:picasso	(file1, -)
:guernica	(file1, -), (file4, -)
:reinasofia	(file1, -)
:rodin	(file2, -)
:museerodin	(file2, -)
:firstName	(file1, -), (file2,-), (file5, -)
:paints	(file1,-)
…	…

```
SELECT ?y
WHERE {?x :firstName ?y .
       ?x :paints :guernica .}
```

Lookups



S3

file1.rdf
```
:picasso type :cubist .
:picasso :firstName "Pablo" .
…
```

file2.rdf
```
:rodin type :sculptor .
:rodin :firstName "Auguste" .
…
```

file3.rdf
```
:reinasofia :located :madrid.
:museerodin :located :paris .
…
```

file4.rdf
```
:guernica :exhibited :reinasofia.
:thethinker :exhibited :museerodin.
…
```

file5.rdf
```
:cubist sc :painter.
:firstName domain :artist .
…
```

Attribute-based example

S table

key	(attribute, value)
:picasso	(file1, -)
:guernica	(file4, -)
:reinasofia	(file1, -)
:rodin	(file2, -)
:thethinker	(file2, -)
:firstName	(file4, -)
…	…

P table

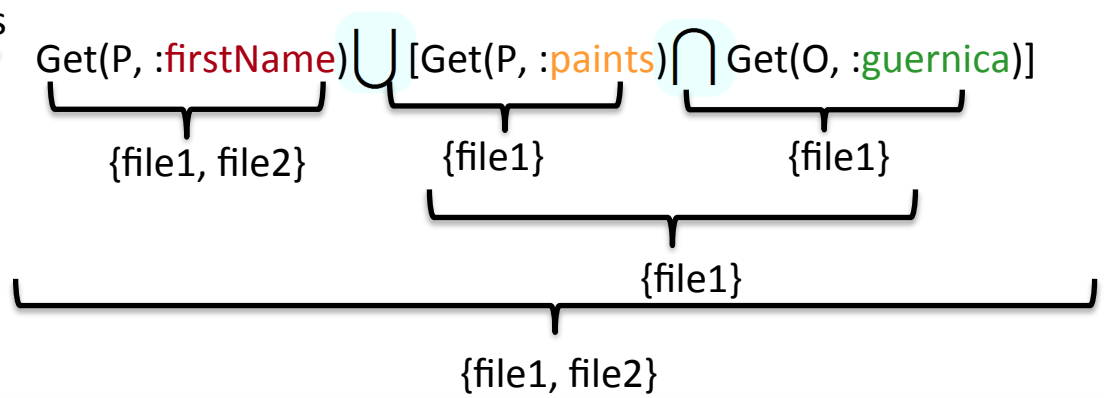
key	(attribute, value)
type	(file1,-), (file3, -), (file4, -)
:firstName	(file1,-), (file2, -)
:paints	(file1,-)
:exhibitedIn	(file1,-), (file2, -)
:locatedIn	(file1,-)
…	…

O table

key	(attribute, value)
:cubist	(file1,-)
:guernica	(file1,-)
"Pablo"	(file1,-)
:sculptor	(file2,-)
:thethinker	(file2,-)
…	…

```
SELECT ?y
WHERE {?x :firstName ?y .
       ?x :paints :guernica .}
```

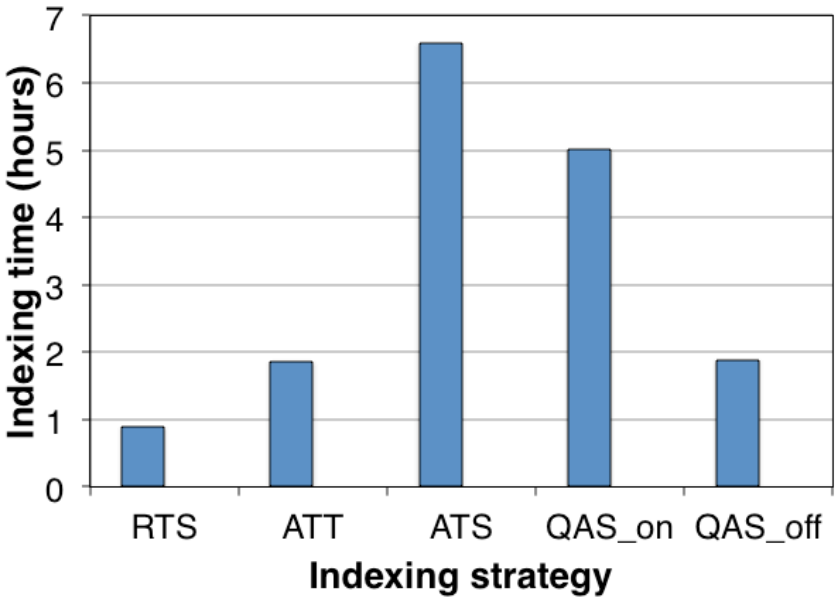
Lookups

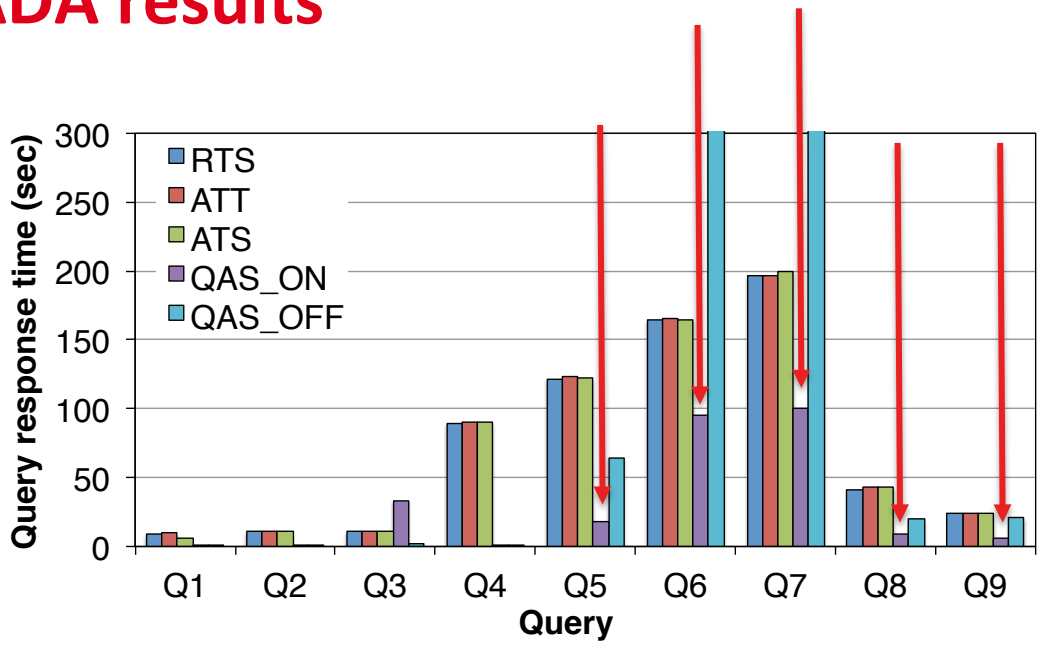


S3

file1.rdf	file2.rdf	file3.rdf	file4.rdf	file5.rdf
:picasso type :cubist . :picasso :firstName "Pablo" . … ✔	:rodin type :sculptor . :rodin :firstName "Auguste" . … ✔	:reinasofia :located :madrid. :museerodin :located :paris . …	:guernica :exhibited :reinasofia. :thethinker :exhibited :museerodin. …	:cubist sc :painter. :firstName domain :artist . …

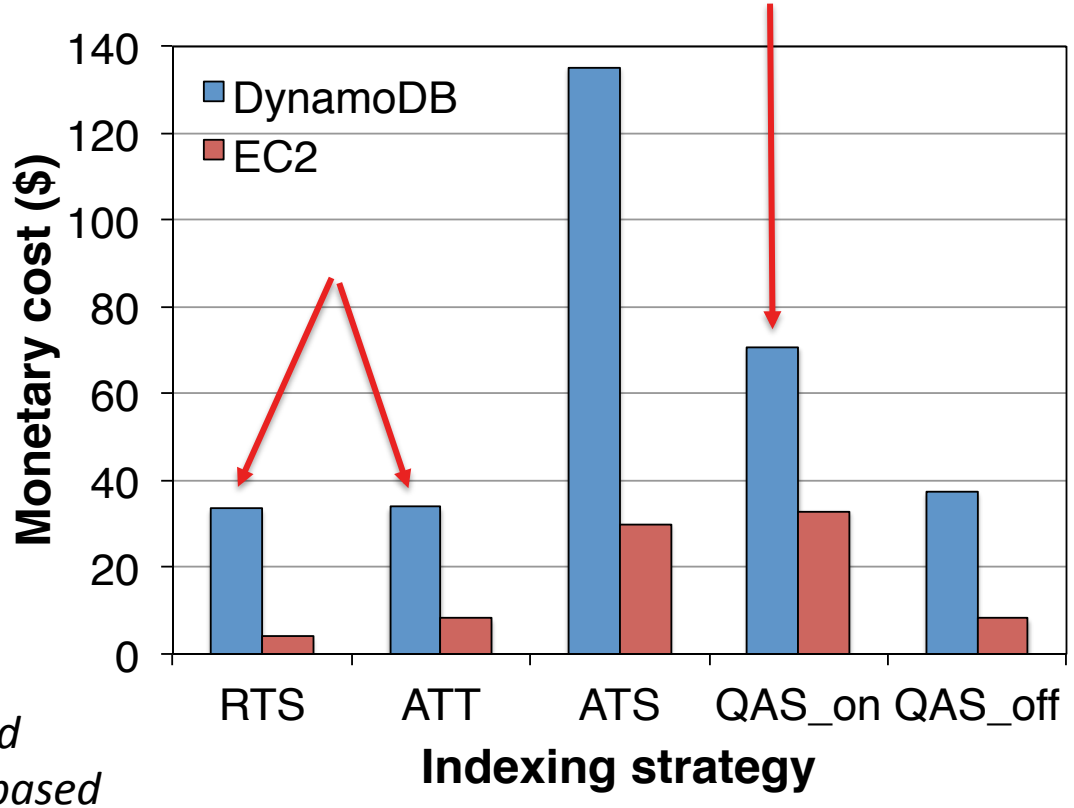
AMADA results





RTS: *term-based*
ATT: *attribute-based*
ATS: *attribute-subset*
QAS: *answering from the index*
on: *w compression*
off: *w/o compression*

What about the money spent?

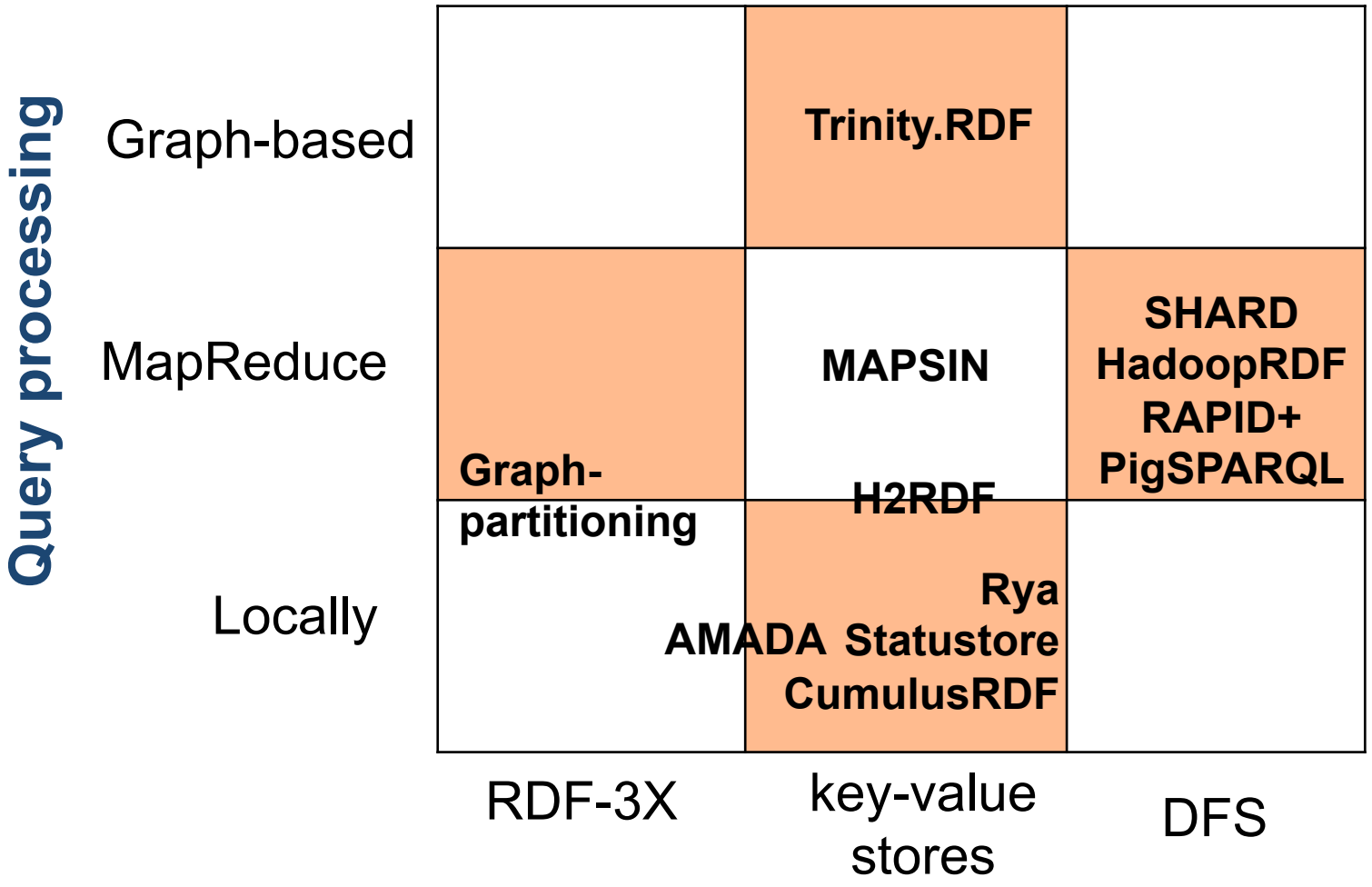


RTS: *term-based*
ATT: *attribute-based*
ATS: *attribute-subset*
QAS: *answering from the index*
on: *w compression*
off: *w/o compression*

IV. RDF stores based on cloud services

Pros	Cons
Scalability	Depends on data partitioning
Fault-tolerance	Separation between storage and processing services
Elastic allocation of resources	Low latency for non-selective queries
Inter-query parallelism	No control on the services
Benefit from higher-level services (PaaS)	

Classification of systems



Analysis dimensions

1. Data storage

- Key-value stores
- DFS (distributed file system)
- Single-site RDF stores or data storage services supplied by cloud providers

2. Query processing

- MapReduce-based
- Local processing (joins are usually performed at a single site)
- Distributed graph-based exploration

3. RDFS entailment

Analysis dimensions

1. Data storage

- Key-value stores
- DFS (distributed file system)
- Single-site RDF stores or data storage services supplied by cloud providers

2. Query processing

- MapReduce-based
- Local processing (joins are usually performed at a single site)
- Distributed graph-based exploration

3. RDFS entailment

Categorization based on RDFS entailment

I. Computing RDFS closure

II. Approaches based on query reformulation

III. Hybrid approaches

I. Computing RDFS closure in the cloud

- RDFS closure computation in MapReduce Hadoop: **WebPie** [Urbani09]
  - RDF data is stored in HDFS
  - Compute all entailed triples using MapReduce
  - Optimization techniques

- Full RDFS closure computation in parallel (MPI-based) [Weaver09]
  - embarrassingly parallel algorithm
  - RDFS triples are kept in memory in each process

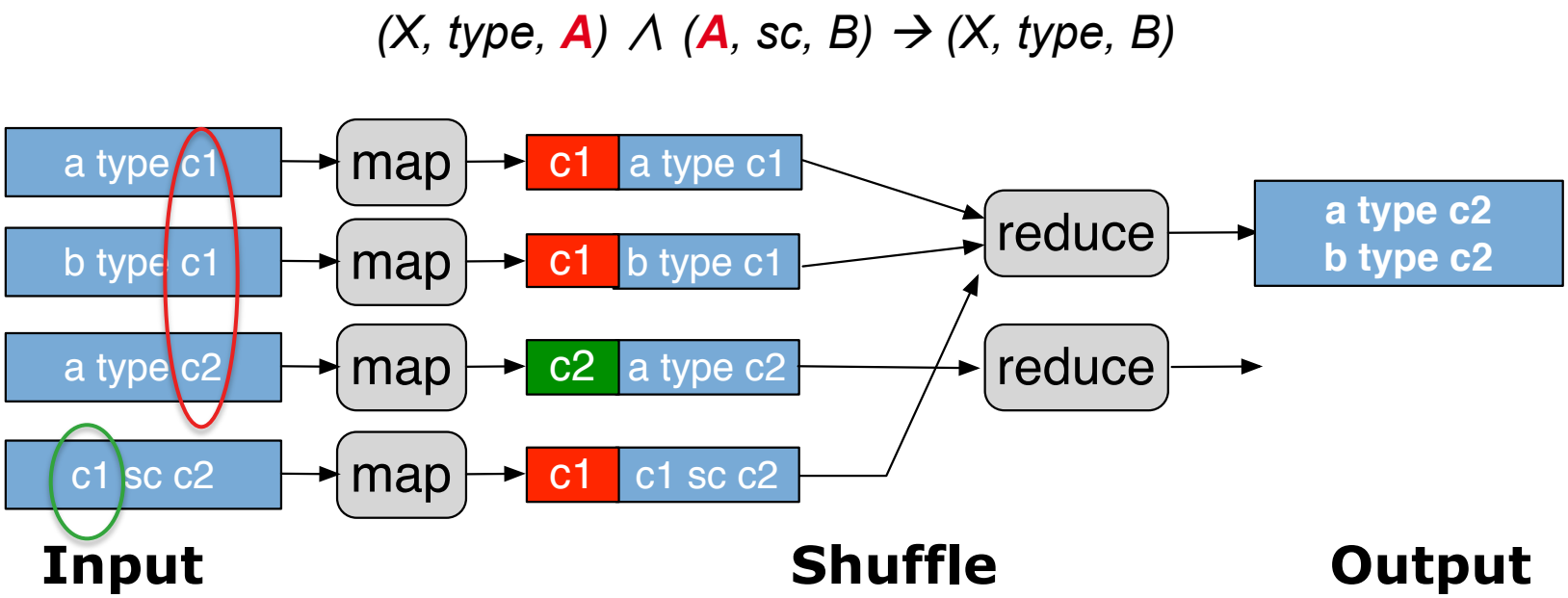
How do we execute the RDFS rules in MapReduce?

Rule Name	If E contains:	then add:
rdfs1	uuu aaa lll. where lll is a plain literal (with or without a language tag).	_:nnn rdf:type rdfs:Literal . where _:nnn identifies a blank node allocated to lll by rule rule lg.
rdfs2	aaa rdfs:domain xxx . uuu aaa yyy .	uuu rdf:type xxx .
rdfs3	aaa rdfs:range xxx . uuu aaa vvv .	vvv rdf:type xxx .
rdfs4a	uuu aaa xxx .	
rdfs4b	uuu aaa vvv.	
rdfs5	uuu rdfs:subPropertyOf vvv . vvv rdfs:subPropertyOf xxx .	
rdfs6	uuu rdf:type rdf:Property .	
rdfs7	aaa rdfs:subPropertyOf bbb . uuu aaa yyy .	
rdfs8	uuu rdf:type rdfs:Class .	uuu rdfs:subClassOf rdfs:Resource .
rdfs9	uuu rdfs:subClassOf xxx . vvv rdf:type uuu .	vvv rdf:type xxx .
rdfs10	uuu rdf:type rdfs:Class .	uuu rdfs:subClassOf uuu .
rdfs11	uuu rdfs:subClassOf vvv . vvv rdfs:subClassOf xxx .	uuu rdfs:subClassOf xxx .
rdfs12	uuu rdf:type rdfs:ContainerMembershipProperty .	uuu rdfs:subPropertyOf rdfs:member .
rdfs13	uuu rdf:type rdfs:Datatype .	uuu rdfs:subClassOf rdfs:Literal .

There is a common constant for rules with 2 atoms → Join

RDFS closure in MapReduce [Urbani09]

- Implement as a join in MapReduce each rule that has two atoms in its body



How do we execute the RDFS rules in MapReduce?

Rule Name	If E contains:	then add:
rdfs1	uuu aaa lll. where lll is a plain literal (with or without a language tag).	_:nnn rdf:type rdfs:Literal . where _:nnn identifies a blank node allocated to lll by rule rule lg.
rdfs2	aaa rdfs:domain xxx . uuu aaa yyy .	uuu rdf:type xxx .
rdfs3	aaa rdfs:range xxx . uuu aaa vvv .	vvv rdf:type xxx .
rdfs4a	uuu aaa xxx .	
rdfs4b	uuu aaa vvv.	
rdfs5	uuu rdfs:subPropertyOf vvv vvv rdfs:subPropertyOf xxx .	
rdfs6	uuu rdf:type rdf:Property	
rdfs7	aaa rdfs:subPropertyOf bbb uuu aaa yyy .	
rdfs8	uuu rdf:type rdfs:Class .	
rdfs9	uuu rdfs:subClassOf xxx . vvv rdf:type uuu .	vvv rdf:type xxx .
rdfs10	uuu rdf:type rdfs:Class .	uuu rdfs:subClassOf uuu .
rdfs11	uuu rdfs:subClassOf vvv . vvv rdfs:subClassOf xxx .	uuu rdfs:subClassOf xxx .
rdfs12	uuu rdf:type rdfs:ContainerMembershipProperty .	uuu rdfs:subPropertyOf rdfs:member .
rdfs13	uuu rdf:type rdfs:Datatype .	uuu rdfs:subClassOf rdfs:Literal .

Entailed triples are used as input in the rules → iterate jobs until no new data is produced (fixpoint)

RDFS closure in MapReduce [Urbani09]

- Optimizations

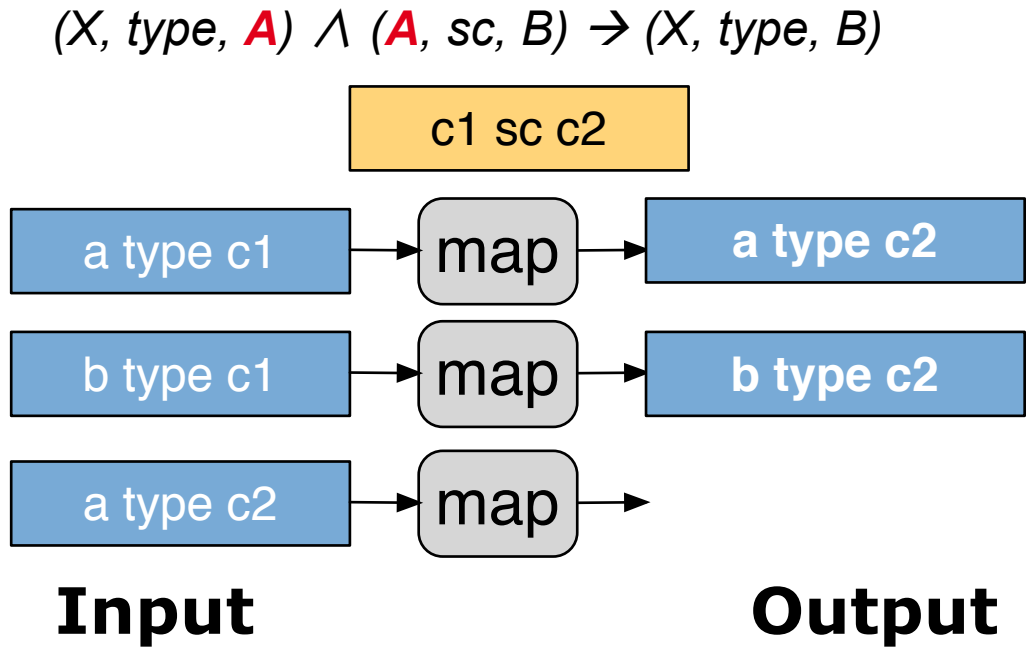
RDFS rules

Rule Name	If E contains:	then add:
rdfs1	uuu aaa lll. where lll is a plain literal (with or without a language tag).	_:nnn rdf:type rdfs:Literal . where _:nnn identifies a blank node allocated to lll by rule rule lg.
rdfs2	aaa rdfs:domain xxx . uuu aaa yyy .	uuu rdf:type xxx .
rdfs3	aaa rdfs:range xxx . uuu aaa vvv .	vvv rdf:type xxx .
rdfs4a	uuu aaa xxx .	uuu rdf:type rdfs:Resource .
rdfs4b	uuu aaa vvv.	vvv rdf:type rdfs:Resource .
rdfs5	uuu rdfs:subPropertyOf vvv . vvv rdfs:subPropertyOf xxx .	
rdfs6	uuu rdf:type rdf:Property .	
rdfs7	aaa rdfs:subPropertyOf bbb . uuu aaa yyy .	
rdfs8	uuu rdf:type rdfs:Class .	uuu rdfs:subClassOf rdfs:Resource .
rdfs9	uuu rdfs:subClassOf xxx . vvv rdf:type uuu .	vvv rdf:type xxx .
rdfs10	uuu rdf:type rdfs:Class .	uuu rdfs:subClassOf uuu .
rdfs11	uuu rdfs:subClassOf vvv . vvv rdfs:subClassOf xxx .	uuu rdfs:subClassOf xxx .
rdfs12	uuu rdf:type rdfs:ContainerMembershipProperty .	uuu rdfs:subPropertyOf rdfs:member .
rdfs13	uuu rdf:type rdfs:Datatype .	uuu rdfs:subClassOf rdfs:Literal .

At least 1 of the atoms is a schema triple!

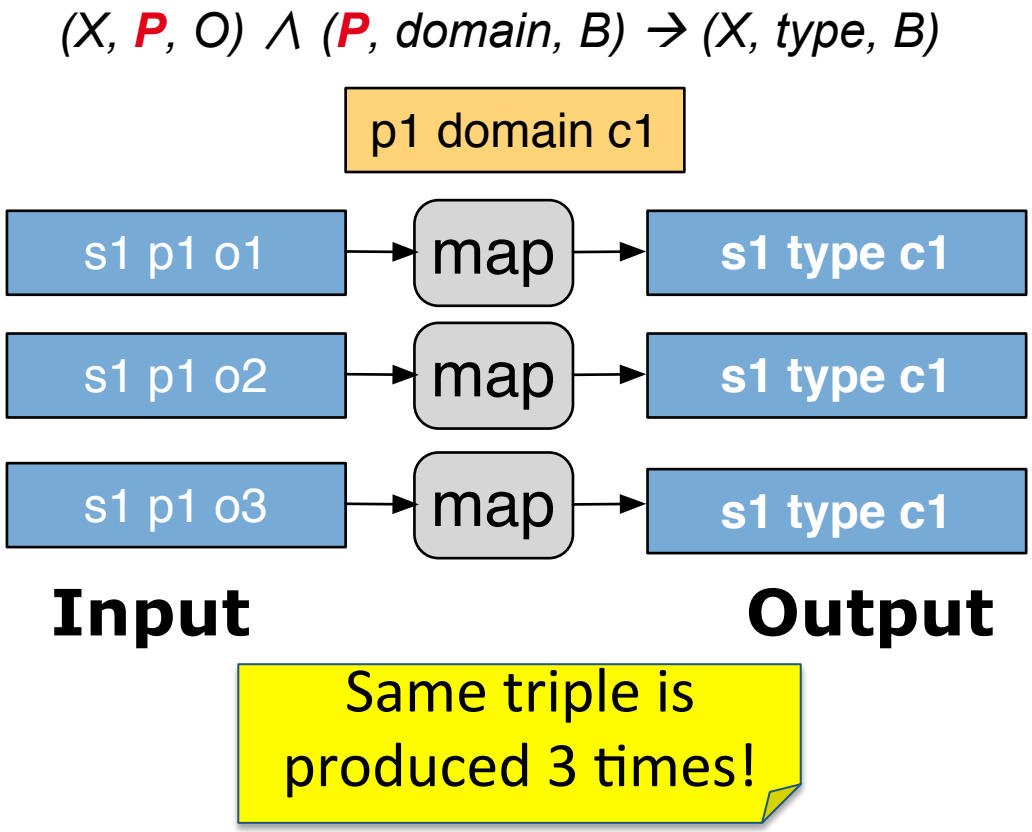
RDFS closure in MapReduce [Urbani09]

- Optimizations
  - Keep the RDF schema triples in memory of each node



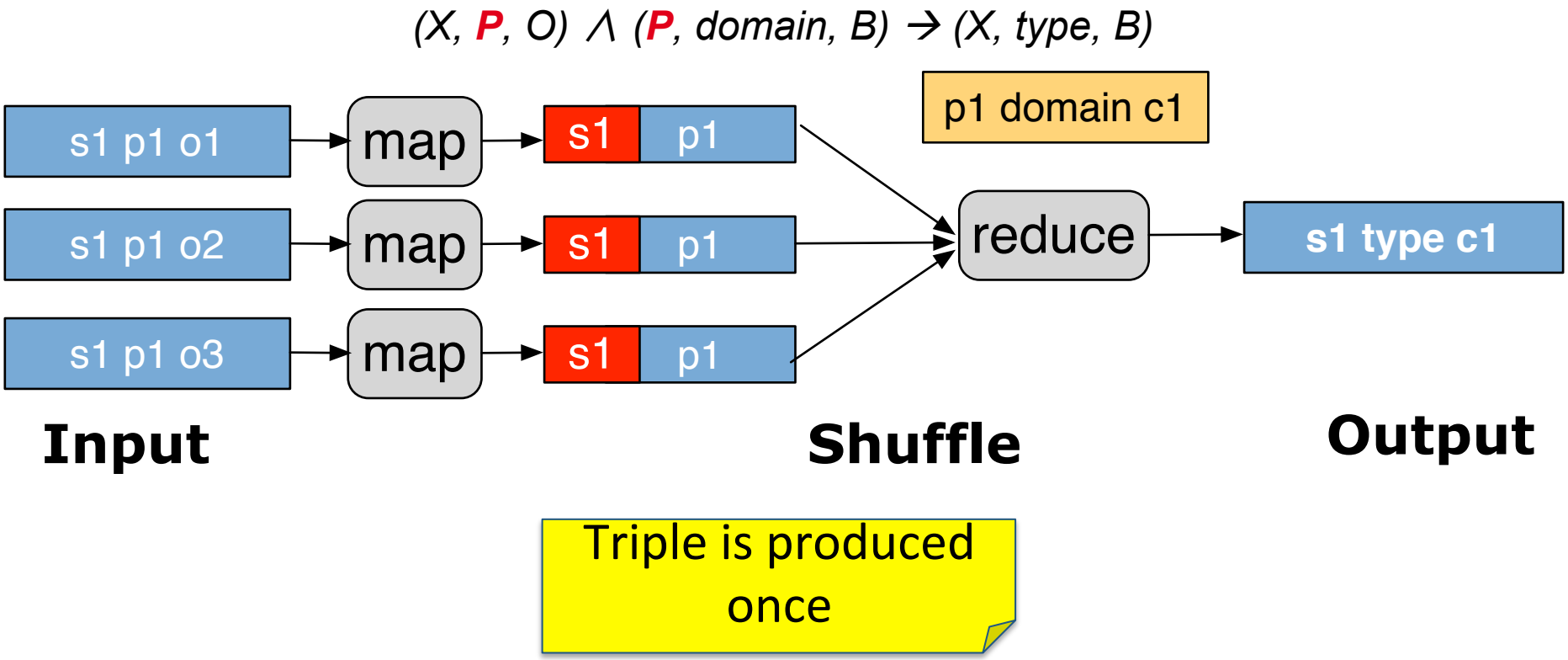
RDFS closure in MapReduce [Urbani09]

- Optimizations
  - Keep the RDF schema triples in memory of each node



RDFS closure in MapReduce [Urbani09]

- Optimizations
  - Keep the RDF schema triples in memory of each node
  - Data grouping to avoid duplicates



RDFS closure in MapReduce [Urbani09]

- Optimizations
  - Keep the RDF schema triples in memory of each node
  - Data grouping to avoid duplicates
  - Ordering of rules to limit iterations

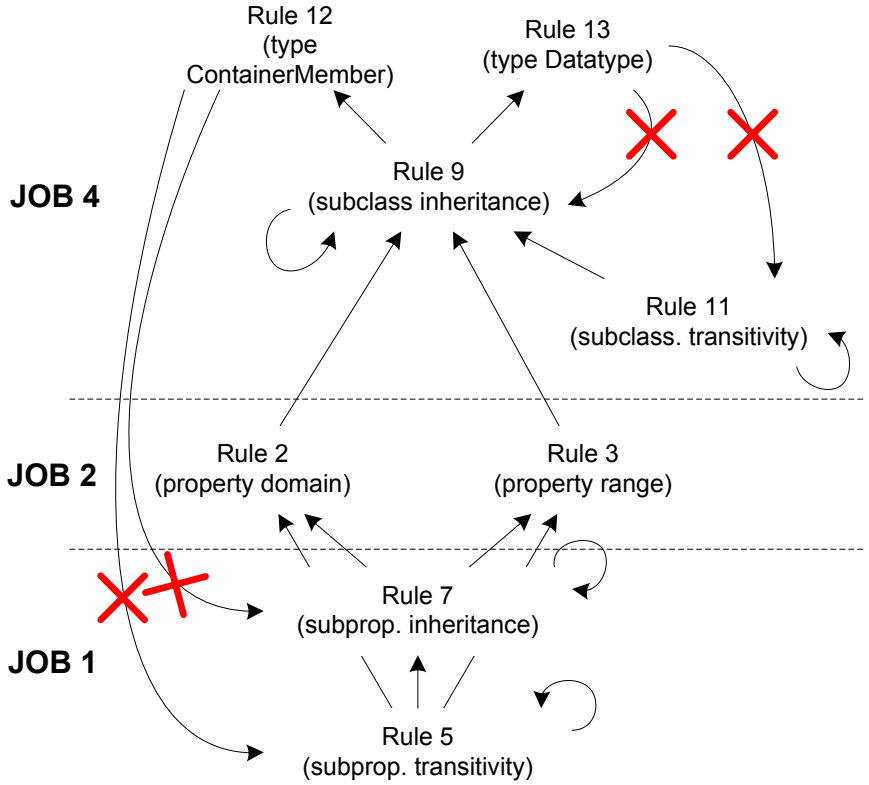


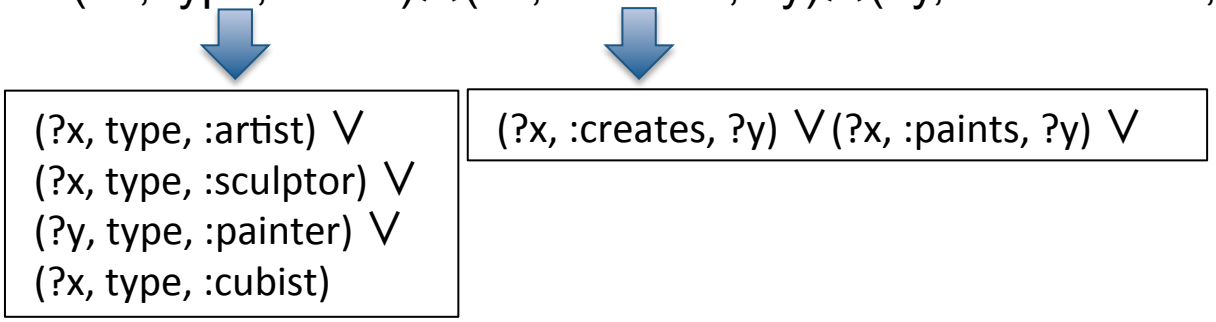
Fig. 3. Relation between the various RDFS rules

Process the rules only once in this order!

II. Query reformulation in the cloud

- Query rewriting of BGP queries may lead to complex queries

$$(?x, type, :artist) \wedge (?x, :creates, ?y) \wedge (?y, :exhibitedIn, ?z)$$



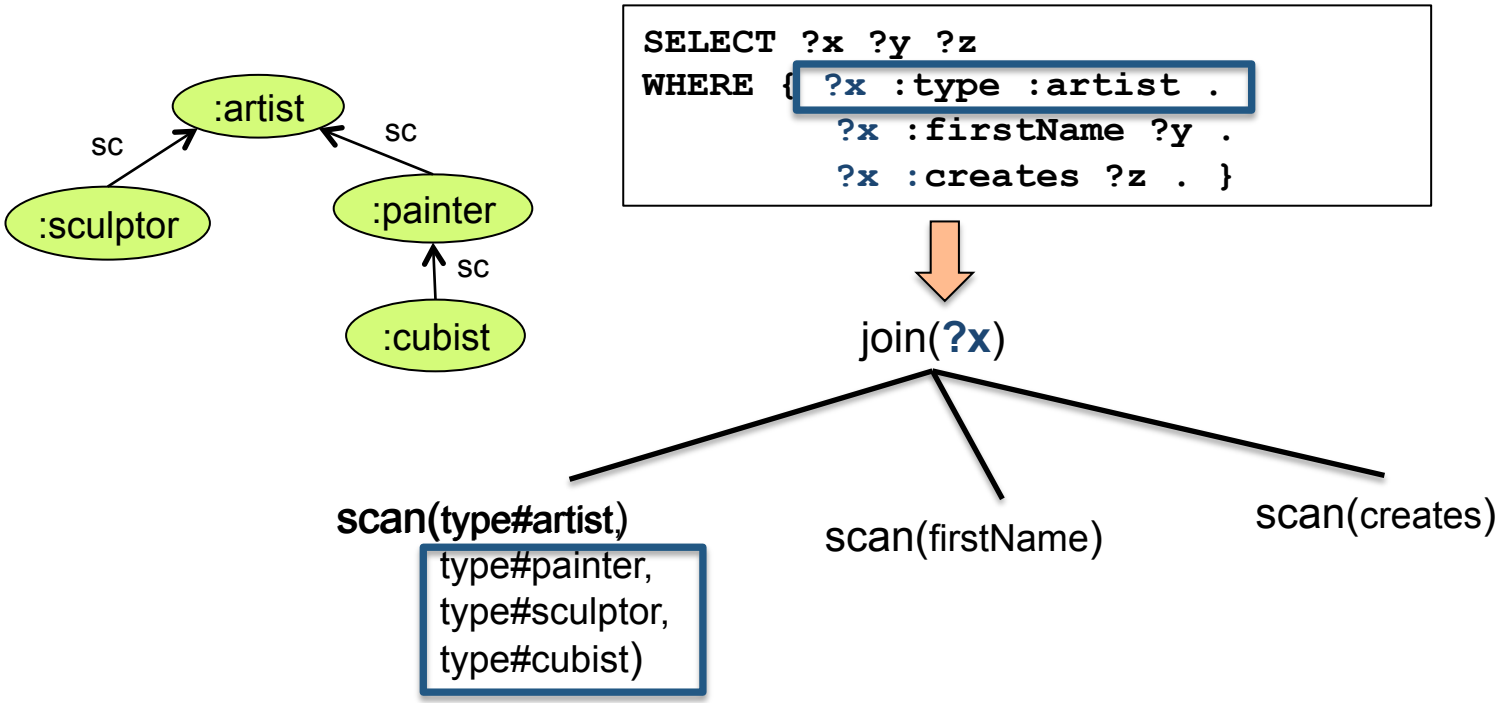
- Conjunction of unions of atomic queries
  - HadoopRDF [Husain11]
- Or union of conjunctive queries?

$$[(?x, type, :artist) \wedge (?x, :creates, ?y) \wedge (?y, :exhibitedIn, ?z)] \vee$$
$$[(?x, type, :artist) \wedge (?x, :paints, ?y) \wedge (?y, :exhibitedIn, ?z)] \vee$$
$$[(?x, type, :sculptor) \wedge (?x, :creates, ?y) \wedge (?y, :exhibitedIn, ?z)] \vee$$
$$[(?x, type, :sculptor) \wedge (?x, :paints, ?y) \wedge (?y, :exhibitedIn, ?z)] \vee$$
$$\ldots\ldots$$

- Transitive closure computation and recursive query processing in MapReduce [Afrati12, Bu12] may apply to RDFS reasoning

Query reformulation in HadoopRDF [Husain11]

- Query reformulation during the file selection of the query processing



III. Hybrid approaches

- Precompute the closure of the schema
  - Schema is usually much smaller than the data
  - Schema does not change often
  - It is always used in the RDFS rules
- Reformulate the query (faster reformulations)
- Representative works
  - QueryPie [Urbani11]
  - Rya [Punnoose12]

Analysis dimensions

1. Data storage

- Key-value stores
- DFS (distributed file system)
- Single-site RDF stores or data storage services supplied by cloud providers

2. Query processing

- MapReduce-based
- Local processing (joins are usually performed at a single site)
- Distributed graph-based exploration

3. RDFS entailment

- Materialization of RDFS closure
- Query reformulation
- Hybrid (only the schema closure is precomputed)

Do we have a winner?

- No, at least not yet ☺
- Performance depends on query workload
  - Star queries or path queries?
  - Selective or analytics-style queries?
- Benchmarks on how real SPARQL queries look:
  - [Duan11]
  - [Arias11]
  - [Picalausa11]
  - Linked Data benchmark council (ldbc.eu)

4

OPEN ISSUES

Where we go from here?

- A lot of room for optimization
  - Improve query response time in the first place
  - Can we apply traditional optimization techniques (join ordering, statistics, etc.)?
- RDF data partitioning
  - Do the existing graph partitioning algorithms fit?
- RDFS entailment
  - Query reformulation
- More SPARQL features
  - Blank nodes, optional, negation, property paths, etc.

Where we go from here?

- RDF views/indexes in the cloud
- SPARQL multi-query optimization in the cloud
- RDF updates in the cloud
- RDF data analytics
- How does cloud variability affect RDF data management?

Thank you

Questions?

http://pages.saclay.inria.fr/zoi.kaoudi/

References

- [Abadi07] D. J. Abadi, A. Marcus, S. Madden, K. J. Hollenbach, “Scalable Semantic Web Data Management Using Vertical Partitioning”, in VLDB 2007.
- [Afrati10] F. N. Afrati and J. D. Ullman, “Optimizing Joins in a Map-Reduce Environment,” in EDBT 2010.
- [Afrati11a] F. N. Afrati and J. D. Ullman, “Optimizing Multiway Joins in a Map-Reduce Environment,” IEEE Trans. Knowl. Data Eng., 2011.
- [Afrati11b] F. N. Afrati, V. R. Borkar, M. J. Carey, N. Polyzotis, and J. D. Ullman, “Map-Reduce Extensions and Recursive Queries,” in EDBT 2011.
- [Afrati12] F. N. Afrati, J. D. Ullman, “Transitive closure and recursive Datalog implemented on clusters” in EDBT 2012.
- [Alexaki01] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis. “On Storing Voluminous RDF Descriptions: The Case of Web Portal Catalogs”, in WebDB 2001.
- [Aranda12] A. Aranda-Andujar, F. Bugiotti, J. Camacho-Rodriguez, D. Colazzo, F. Goasdoué, Z. Kaoudi, and I. Manolescu, “Amada: Web Data Repositories in the Amazon Cloud (demo)”, in CIKM 2012.
- [Arias11] M. Arias, J. D. Fernandez, M. A. Martínez-Prieto, “An Empirical Study of Real-World SPARQL Queries”, in USEWOD 2011.
- [Battre06] D. Battre, A. Hoing, F. Heine, O. Kao, “On Triple Dissemination, Forward-Chaining, and Load Balancing in DHT based RDF stores”, in DBISP2P 2006.
- [Blanas10] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, “A Comparison of Join Algorithms for Log Processing in MapReduce,” in SIGMOD 2010.

References

- [Bu12] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, “The HaLoop Approach to Large-Scale Iterative Data Analysis,” VLDB J, 2012.
- [Bugiotti12] F. Bugiotti, F. Goasdoué, Z. Kaoudi, and I. Manolescu, “RDF Data Management in the Amazon Cloud,” in ICDT/EDBT Workshops 2012.
- [Cattell11] R. Cattell, “Scalable SQL and NoSQL data stores,” SIGMOD Record, May 2011.
- [Dean13] J. Dean, L. A. Barroso, “Tail at Scale”, communication of the ACM, February 2013.
- [Duan11] S. Duan, A. Kementsietsidis, K. Srinivas, O. Udrea, “Apples and oranges: a comparison of RDF benchmarks and real RDF datasets”, in SIGMOD 2011.
- [Goasdoué13] F. Goasdoué, I. Manolescu, and A. Roatiș, “Efficient Query Answering against Dynamic RDF Databases”, in EDBT 2013.
- [Hayes04] P. Hayes, “RDF Semantics”, W3C Recommendation, February 2004, http://www.w3.org/TR/rdf- mt/.
- [Hose13] K. Hose, and R. Schenkel, “WARP: Workload-Aware Replication and Partitioning for RDF”, in DESWEB 2013.
- [Huang11] J. Huang, D. J. Abadi, and K. Ren, “Scalable SPARQL Querying of Large RDF Graphs,” PVLDB 2011.
- [Husain11] M. Husain, J. McGlothlin, M. M. Masud, L. Khan, and B. M. Thuraisingham, “Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing,” IEEE Trans. on Knowl. and Data Eng., 2011.
- [Iosup11] A. Iosup, N. Yigitbasi, D.H. J. Epema, “On the Performance Variability of Production Cloud Services”, in CCGRID 2011.

References

- [Kaoudi13] Z. Kaoudi, M. Koubarakis, “Distributed RDFS Reasoning over Structured Overlay Networks”, Journal on Data Semantics, 2013.
- [Kotoulas10] S. Kotoulas, E. Oren, F. Harmelen, “Mind the data skew: distributed inferencing by speeddating in elastic regions”, in WWW 2010.
- [Mallea11] A. Mallea, M. Arenas, A. Hogan, A. Polleres, “On Blank Nodes”, in ISWC 2011.
- [METIS] METIS - Serial Graph Partitioning and Fill-reducing Matrix Ordering: http://glaros.dtc.umn.edu/gkhome/metis/metis/overview
- [Neumann10] T. Neumann and G. Weikum, “The RDF-3X Engine for Scalable Management of RDF Data,” VLDBJ 2010.
- [Picalausa11] F. Picalausa, S. Vansummeren, “What are real SPARQL queries like?”, in SWIM 2011.
- [Punnoose12] R. Punnoose, A. Crainiceanu, and D. Rapp, “Rya: A Scalable RDF Triple Store for the Clouds,” in 1st International Workshop on Cloud Intelligence (in conjunction with VLDB), 2012.
- [Ravindra11] P. Ravindra, H. Kim, K. Anyanwu, “An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce”, in ESWC 2011.
- [Rohloff10] K. Rohloff and R. E. Schantz, “High-Performance, Massively Scalable Distributed Systems using the MapReduce Software Framework: the SHARD Triple-Store,” in Programming Support Innovations for Emerging Distributed Applications, 2010.
- [Schad10] J. Schad, J. Dittrich, J. Quiané-Ruiz, “Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance”, PVLDB 2010.

References

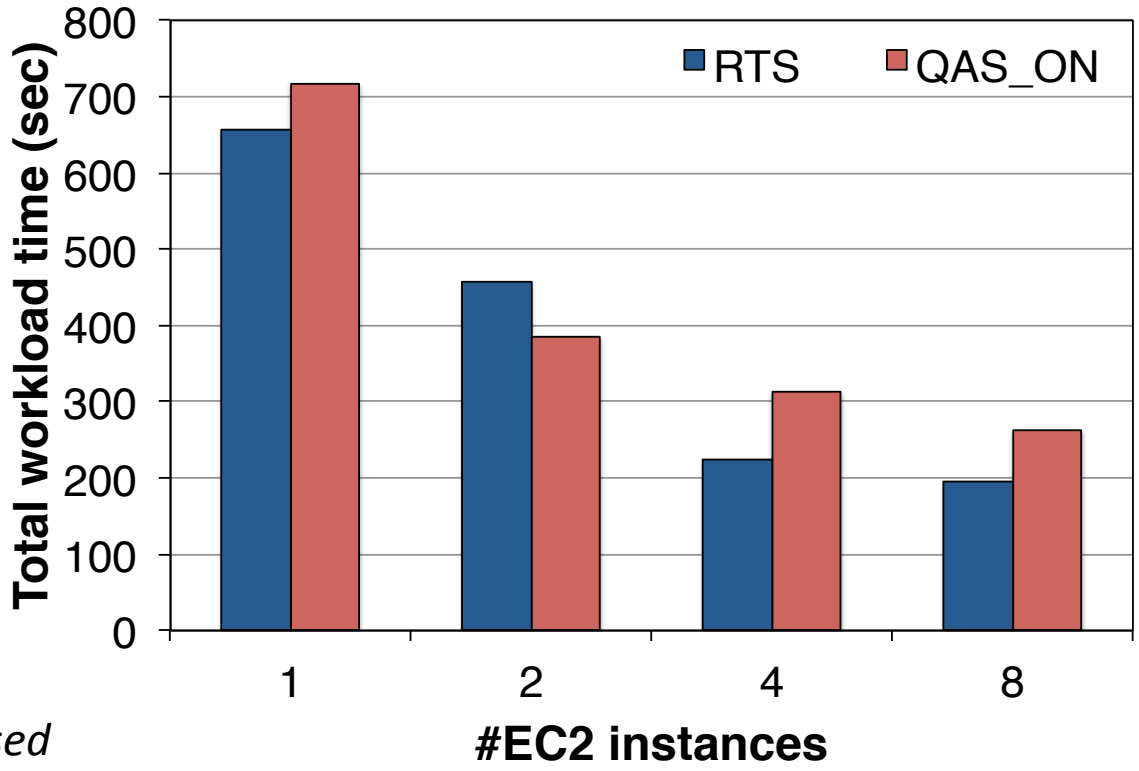
- [Schätzle12] A. Schätzle, M. Przyjaciel-Zablocki, C. Dorner, T. Hornung, G. Lausen, “Cascading Map-Side Joins over HBase for Scalable Join Processing”, in SSWS+HPCSW 2012.
- [Schätzle11] A. Schätzle, M. Przyjaciel-Zablocki, G. Lausen, ”PigSPARQL: mapping SPARQL to Pig Latin”, in SWIM 2011.
- [Stein10] R. Stein and V. Zacharias, “RDF On Cloud Number Nine,” in 4th Workshop on New Forms of Reasoning for the Semantic Web: Scalable and Dynamic, May 2010.
- [Theoharis05] Y. Theoharis, V. Christophides, and G. Karvounarakis. “Benchmarking Database Representations of RDF/S Stores”, in ISWC 2005.
- [Urbani09] J. Urbani, S. Kotoulas, E. Oren, and F. van Harmelen, “Scalable Distributed Reasoning using MapReduce,” in ISWC 2009.
- [Urbani11] J. Urbani, F. van Harmelen, S. Schlobach, and H. Bal, “QueryPIE: Backward Reasoning for OWL Horst over Very Large Knowledge Bases,” in ISWC 2011.
- [Weaver09] J. Weaver and J. A. Hendler, “Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples”, in ISWC 2009.
- [Weiss08] C. Weiss, P. Karras, A. Bernstein, “Hexastore: sextuple indexing for semantic web data management”, PVLDB 2008.
- [Wilkinson06] K. Wilkinson, “Jena Property Table Implementation”, in SSWS 2006.

References

- [Wu11] S. Wu, F. Li, S. Mehrotra, B. C. Ooi, “Query Optimization for Massively Parallel Data Processing”, in SOCC 2011.
- [Zeng13] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang, “A Distributed Graph Engine for Web Scale RDF Data”, in PVLDB 2013.
- [Zhang13] X. Zhang, L. Chein, Y. Tong, and M. Wang, “EAGRE: Towards Scalable I/O Efficient SPARQL Query Evaluation on the Cloud”, in ICDE 2013.

Appendix

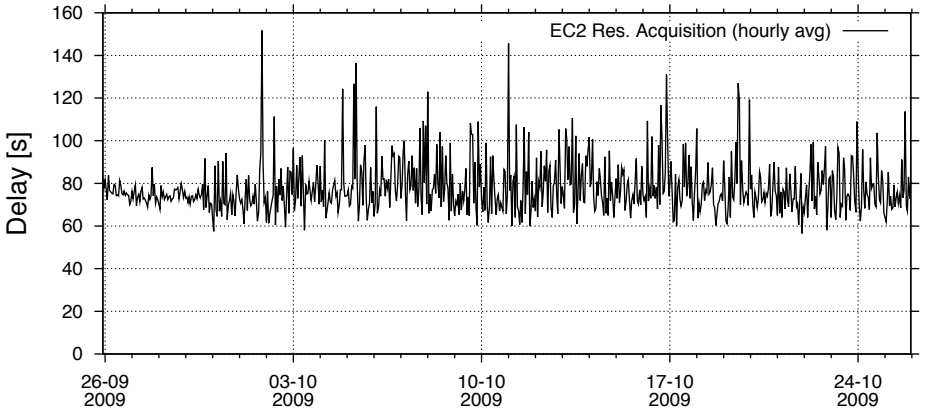
AMADA results



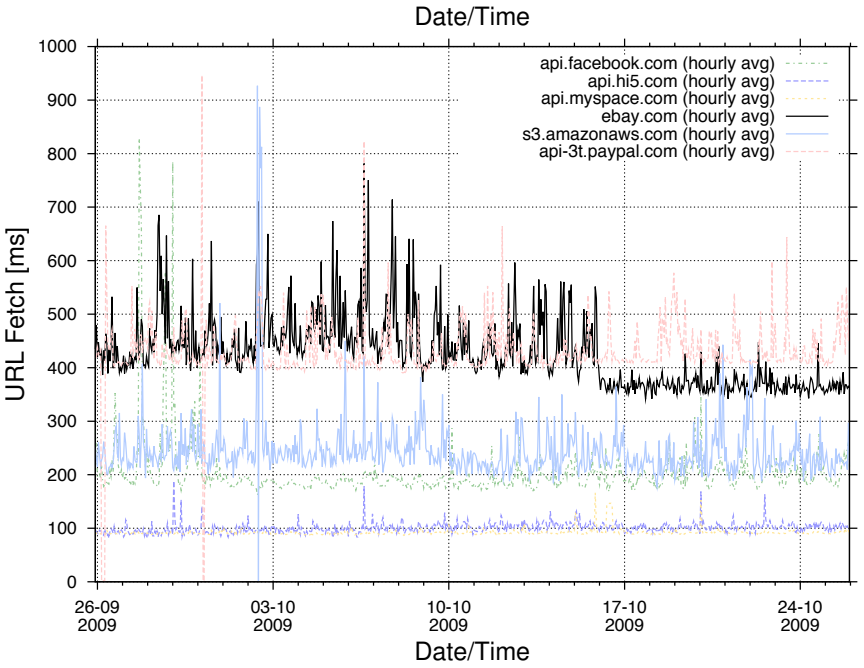
RTS: *term-based*
QAS: *answering from the index*
on: *w compression*

Variability in the clouds

Cloud = performance variability [Schad10], [Iosup11]



from [Iosup11]



from [Iosup11]