Large scale rule-based Reasoning using a Laptop

Martin Peters[1], Sabine Sachweh[1] and Albert Zündorf[2]

[1] University of Applied Sciences and Arts Dortmund, Germany
Smart Environments Engineering Lab

[2] University of Kassel, Germany,
Software Engineering Research Group

Two types of reasoning

- Reasoning is one key feature when using ontologies
- Reasoning means to create new knowledge by inferring facts that are implicitly given by the existing data

- scaleable and fast reasoning can still be a challenging task, depending on
  - the dataset (structure, **number of triples**)
  - the used ontology language

- different ways to perform efficient (and parallel) reasoning:

multicore processors

- fast
- cheap
- limited hardware —> limited number of triples

MapReduce

- fast
- scaleable (WebPie: 100 billion triples)
- expensive

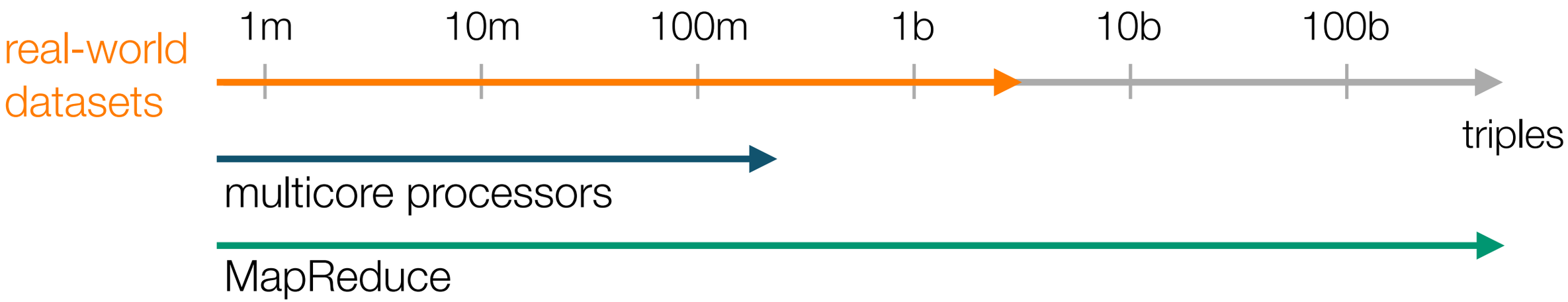
Size of datasets

multicore processors

- fast
- cheap
- limited hardware —> limited number of triples

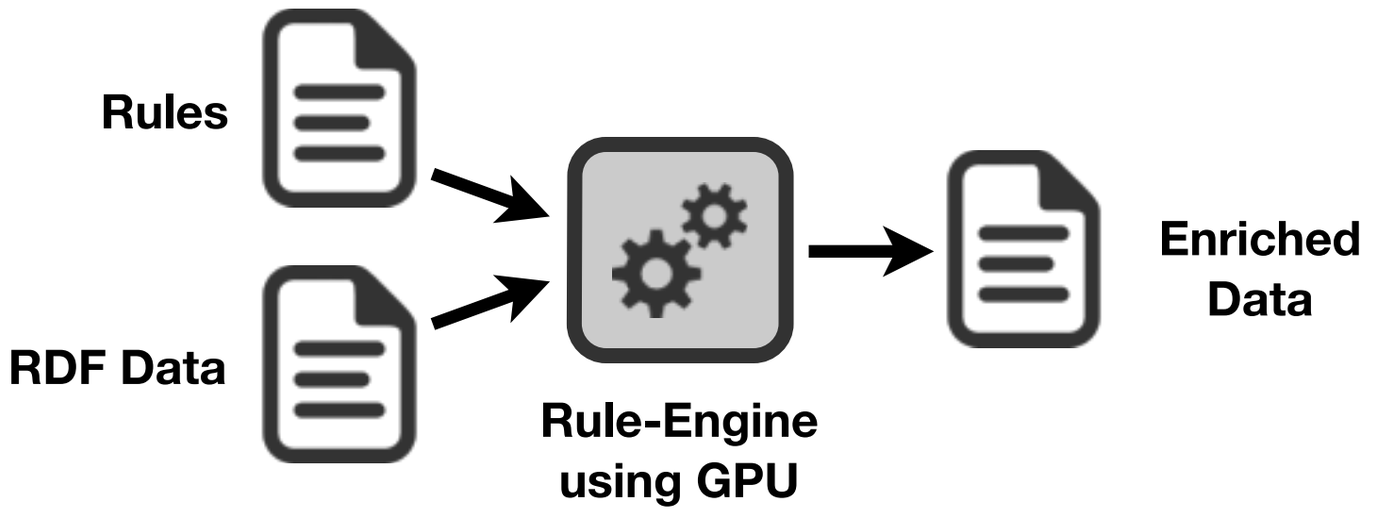
MapReduce

- fast
- scaleable (WebPie: 100 billion triples)
- expensive



Previous work

- **[PBSZ14]: Scaling Parallel Rule-based Reasoning**
  Martin Peters, Christopher Brink, Sabine Sachweh, Albert Zündorf
  11th Extended Semantic Web Conference, 2014
  - a RETE-based forward chaining reasoner implementation that runs on a GPU

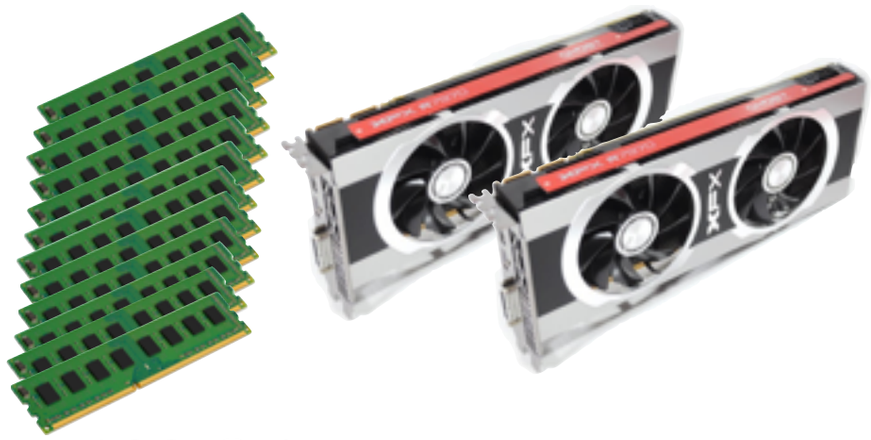


  - allows to define the rules that shall be applied in a simple rule-file
  - **reasoning: 1 billion triples on a server equipped with 2 GPUs and 192 GB memory**

How to do the same reasoning on a laptop?

- limiting factor of the evaluation in [PBSZ14]:
  - **memory consumption**

- **Goal:** identify the resource consuming parts of the RETE-based reasoner process and
  - reduce data that needs to be stored
  - swap data from main memory to the hard disk (without loss of performance)
  - apply compression



192 GB

16 GB

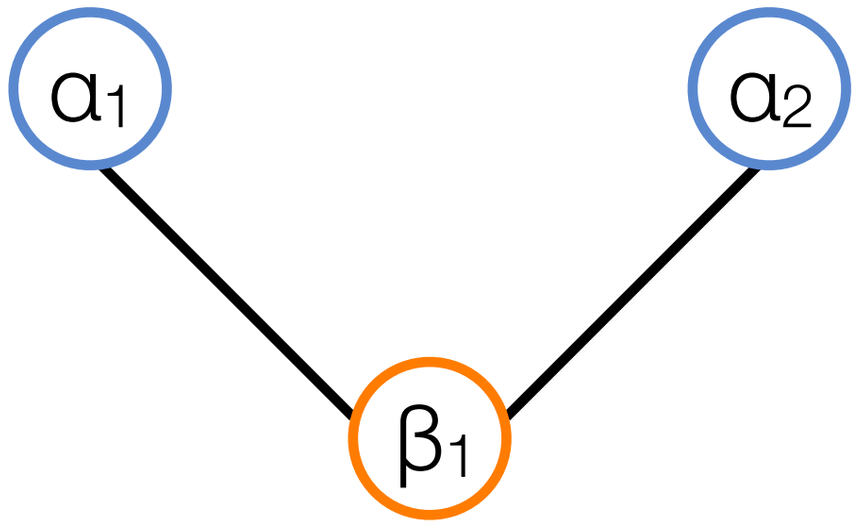
The RETE network

R1: (?x ?p ?y) → (?p rdf:type rdf:Property)

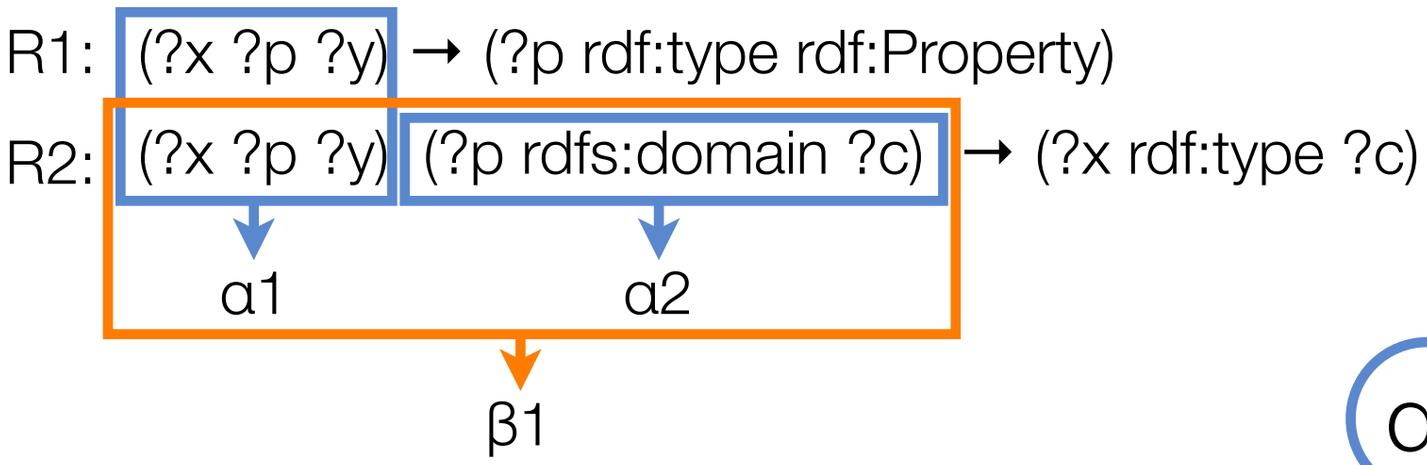
R2: (?x ?p ?y) (?p rdfs:domain ?c) → (?x rdf:type ?c)

α1 α2

β1



alpha- and beta matching



T1: [Bob uni:publishes Paper1]

T2: [Alice uni:publishes Paper2]

T3: [uni:publishes rdfs:domain Researcher]

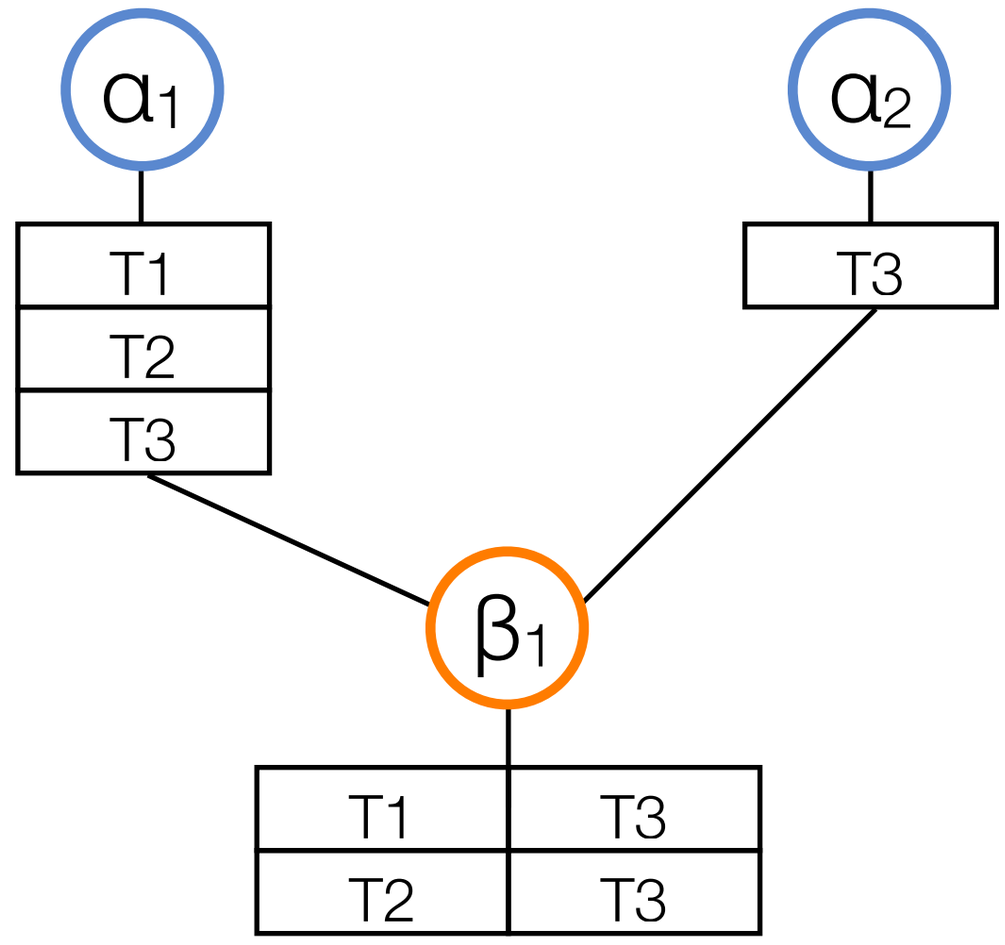
T4: [uni:publishes rdf:type rdf:Property]

[uni:publishes rdf:type rdf:Property]

T5: [rdfs:domain rdf:type rdf:Property]

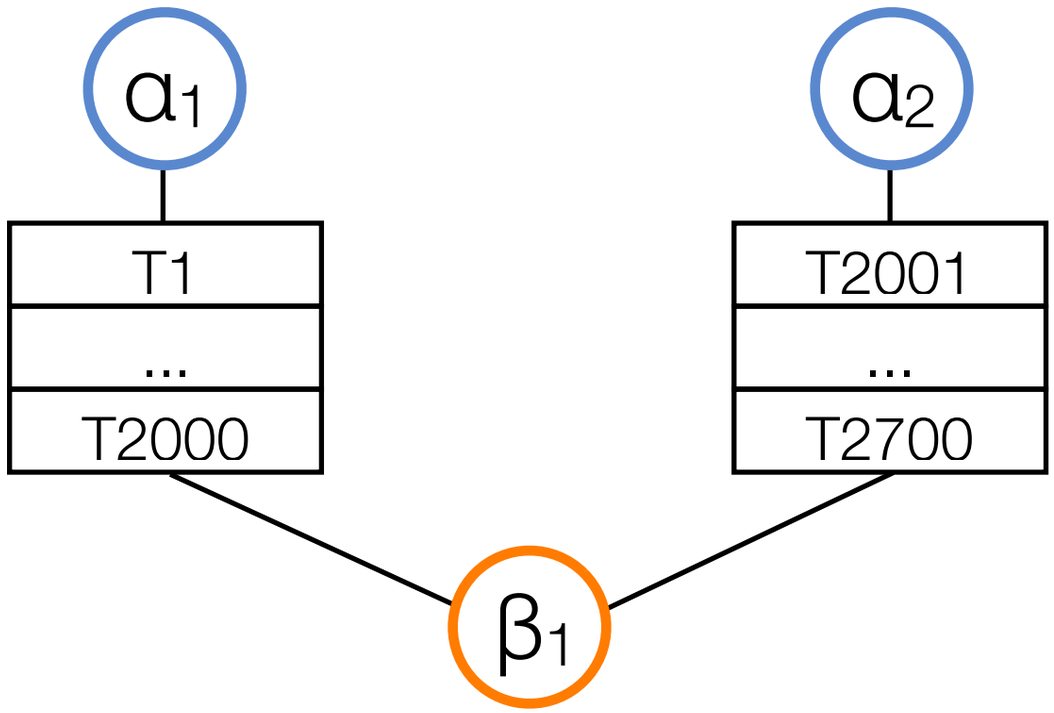
T6: [Bob rdf:type Researcher]

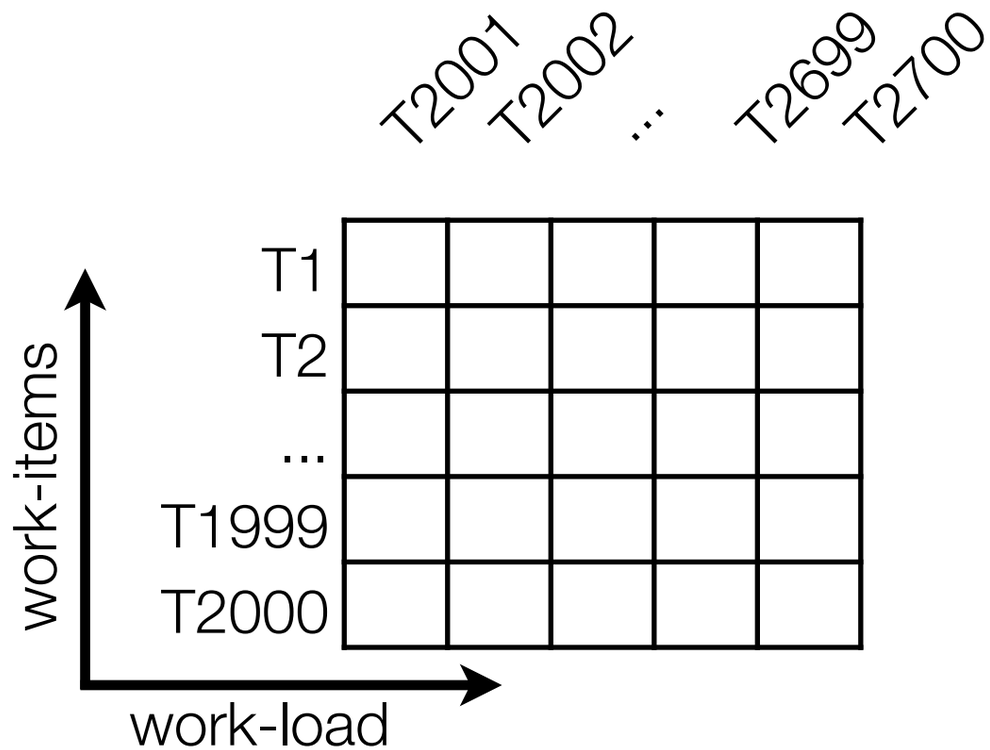
T7: [Alice rdf:type Researcher]



Parallelization for massively parallel hardware

- alpha-matching
  - for each input triple one thread is created that checks, if that triple matches to one or more alpha-nodes (n triples → n threads)
- beta-matching
  - one thread for each match of one of the parent-nodes, that iterates through all matches of the second parent-node and checks for a match





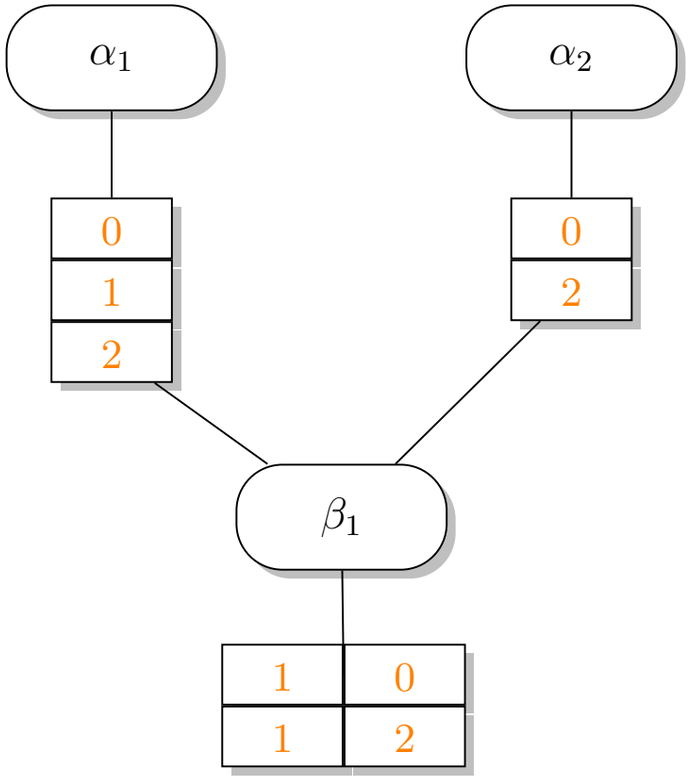
Memory consumption

data structures that need to be stored:

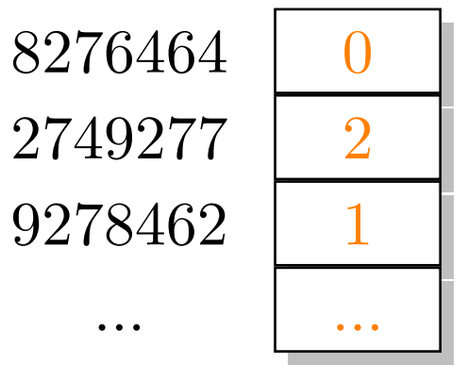
Triples
dictionary-encoded triples

	s	p	o
0	55	79	35
1	22	55	104
2	55	79	82
3	…	…	…

Working memories *W*
of the RETE network



Triple HashSet
to identify duplicate triples



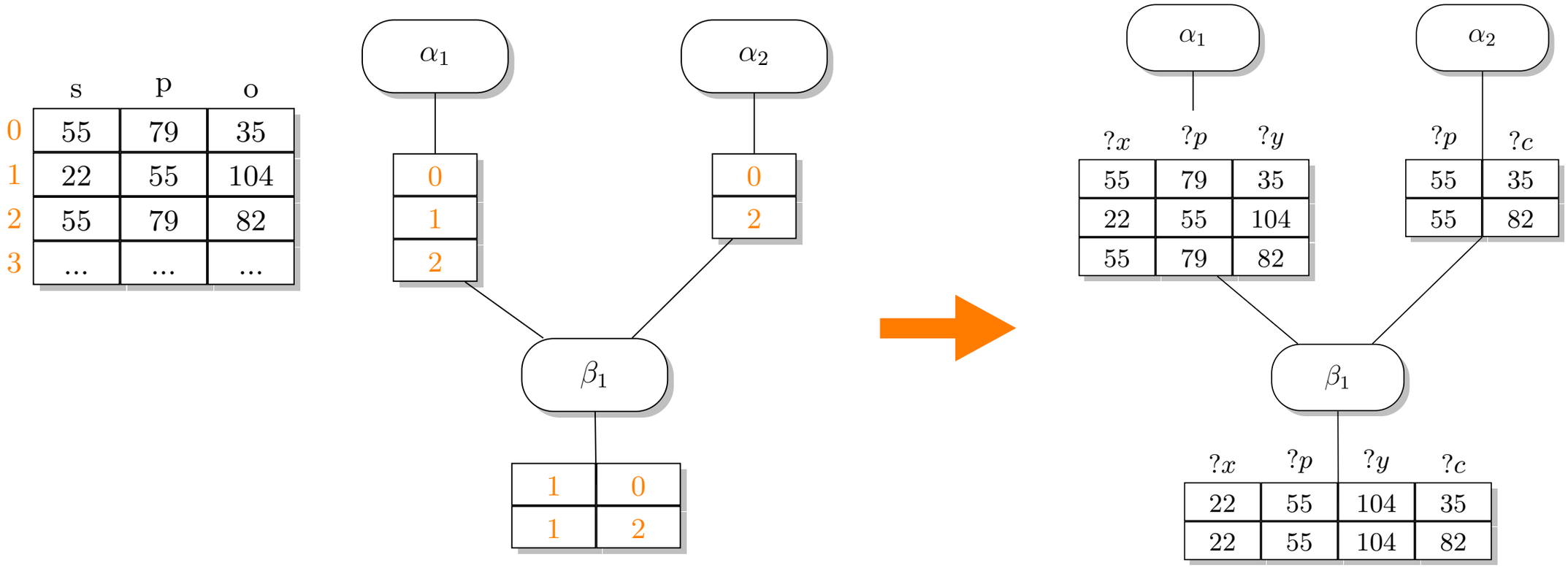
Memory consumption: evaluation

- datasets:
  - Lehigh University Benchmark with 2000 Universities (267M input triples)
  - DBpedia, english version (394M input triples)
- rulesets: RDFS and ρdf (subset of RDFS)

dataset	ruleset	total triples (n)	triple size	triple HashSet	matches	references	size of W	**total size**
LUBM2000	ρdf	333.7 M	8009 MB	3814 MB	287.5 M	1022.5 M	8180 MB	**20.0 GB**
LUBM2000	RDFS	377.1 M	9050 MB	4310 MB	629.1 M	2119.1 M	16953 MB	**30.3 GB**
DBpedia	ρdf	400.6 M	9614 MB	4578 MB	123.1 M	446.7 M	3574 MB	**17.8 GB**
DBpedia	RDFS	475.1 M	11402 MB	5430 MB	554.1 M	1978.5 M	15828 MB	**32.7 GB**

1. Working memories

- references in the working memories are replaced with the actual data
- no references need to be resolved anymore
- working memories can easily be swapped to the hard disk
  (and can efficiently be read in blocks)



old: using references

new: using the actual data

2. Triple HashSet

- is randomly accessed to identify duplicate triples and thus needs to be **kept in main memory**

→ remove triple-references and apply compression

- first step: remove triple references

hashcode	triple-reference
8276464	0
2749277	2
9278462	1
…	…

	s	p	o
0	55	79	35
1	22	55	104
2	55	79	82
3	…	…	…

hashcode	triple		
8276464	55	79	35
2749277	22	55	104
9278462	55	79	82
…	…	…	…

2. Triple HashSet: compression

- **compression first step:** vertical partitioning
  - many datasets consists of only a couple of predicates (LUBM: 43, DBpedia: ~ 53.000)
  - one HashSet for each predicate
  - no need to explicit store the predicate
  - possible memory reduction of ~ 33%

hashcode	s	**p**	o
8276464	55	79	35
2749277	22	55	104
9278462	55	79	82
...	...	...	...

predicate: 79

8276464	55	35
9278462	55	82
...	...	...

predicate: 55

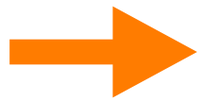
2749277	22	104
...	...	...

2. Triple HashSet: compression

- **compression second step:** differential encoding
  - store only the difference of the first value and the second value, if this is a smaller amount than the value itself
  - this may cause a change of the order of the elements

predicate: 79

8276464	55	35
9278462	55	82
…	…	…



predicate: 79

8276464	55	20
9278462	82	27
…	…	…

2. Triple HashSet: compression

- **compression third step:** variable byte encoding
  - using 8 byte datatypes, many bytes are not used
  - the actual values can be represented using much less bytes

predicate: 79

8276464	55	20
9278462	82	27
…	…	…

55 binary: 0000 … | 0000 0000 | 0011 0111
byte 7 - 2 | byte 1 | byte 0

20 binary: 0000 … | 0000 0000 | 0001 0100
byte 7 - 2 | byte 1 | byte 0

- combine both values:

1011 0111 1001 0100
55 20

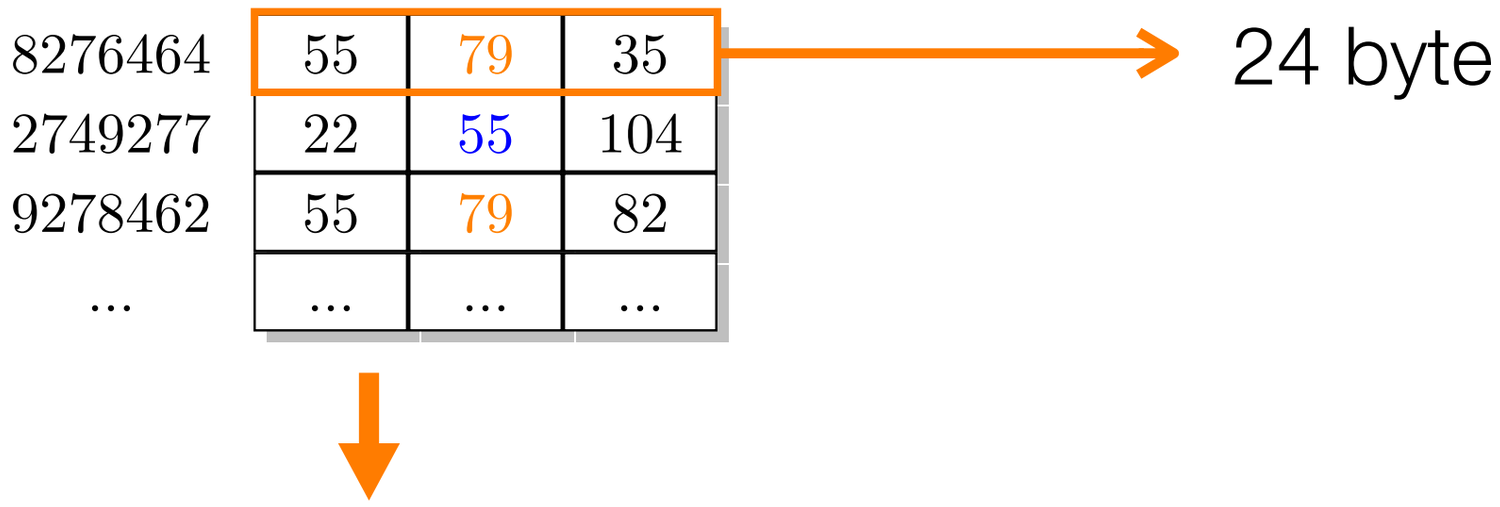
indicates differential encoding

indicates the order of encoded values

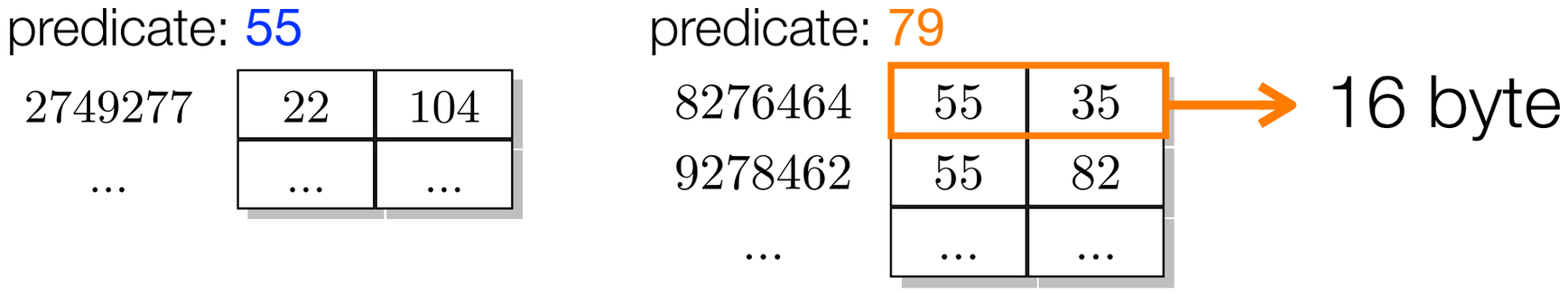
1 indicates end of first data value, **0** otherwise

2. Triple HashSet: compression - summary

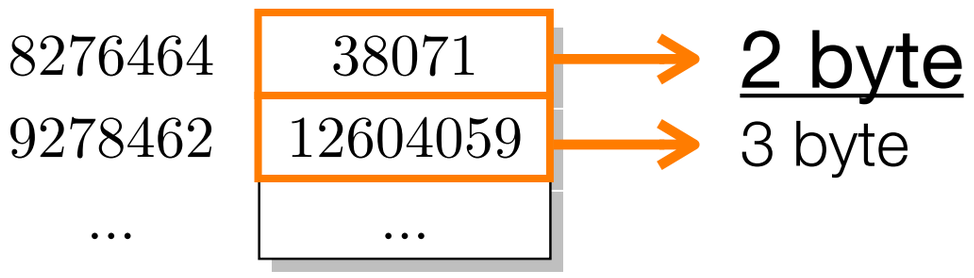
- initial Triple HashSet:



- vertical partitioning:



- differential encoding and variable byte encoding:



Resulting memory consumption

1. **Working memories:** → swapped to the hard disk
2. **Triple HashSet:** → compressed
3. **Triples:** no more need for random access, neither by
   - working memories nor by
   - triple HashSet

→ swapped to the hard disk

but: high complexity for the code that gets executed on the GPU because of new working memories

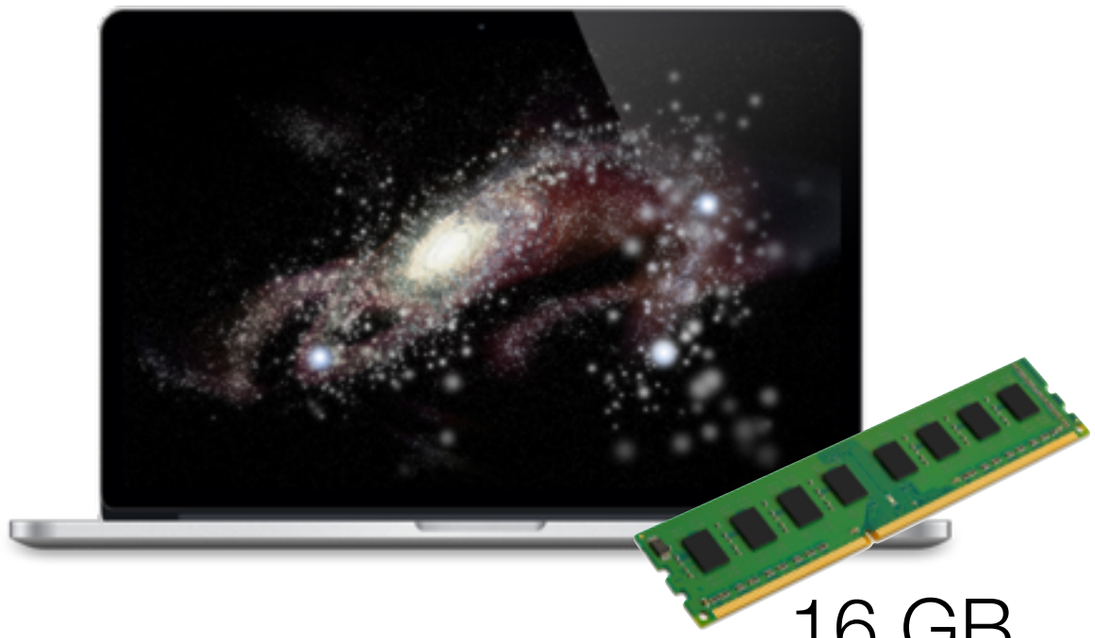
Code generation for the GPU

- Based on the input-rules, the code that is executed on the GPU is generated
  - allows to reduce loops
  - allows to optimize memory access
  - allows to reduce method parameters
  - allows to use dictionary encoded values directly in the code
  - allows to provide a specific, optimized method for each RETE node

```
__kernel void alpha_match_count(...) {
   ...

   // Load only data that is needed
   long triple_p = triples[gid*3+1];

   if(triple_p == 530) {      // alpha node (?a rdfs:subPropertyOf ?b)
      ... // process match
   }

   if(triple_p == 40) {       // alpha node (?a rdfs:subClassOf ?b)
      ... // process match
   }
   ...
```

Test environment

- datasets:
  - Lehigh University Benchmark (LUBM): LUBM1000 to LUBM8000
  - DBpedia 3.9, Englisch version
  - Comparative Toxicogenomics Database (CTD), real world dataset
- ruleset: RDFS and ρdf
- Apple MacBook Retina laptop (2012)
  - 16GB of memory
  - 2.3 GHz Intel Core i7 processor
  - 256GB SSD hard disk
  - NVIDIA GeForce GT 650M graphic card with 1024MB



Memory consumption

- for applying pdf reasoning

dataset	predicates	byte / triple	byte / triple with overhead	total memory
LUBM1000	32	6.04	9.71	1620 MB
LUBM2000	32	6.11	9.29	3098 MB
LUBM4000	32	5.69	8.26	5513 MB
LUBM8000	32	6.16	8.80	11748 MB
DBpedia	53139	7.45	12.81	5129 MB
CTD	43	5.94	9.36	3137 MB

- byte per triple with overhead includes unused entries in the HashSet
- average byte consumption per triple: **6,2 byte** / 9,7 byte
  - → reduction of about 75% for the triple HashSet
  - → reduction of about 84% for the complete RETE algorithm

Reasoning performance

dataset	input triples	ρdf total triples	ρdf reasoning	ρdf throughput	RDFS total triples	RDFS reasoning	RDFS throughput
LUBM1000	134 M	167 M	41.6 s	4017 ktps	189 M	114.1 s	1653 ktps
LUBM2000	267 M	334 M	98.4 s	3391 ktps	377 M	287.6 s	1312 ktps
LUBM4000	534 M	668 M	296.9 s	2249 ktps	754 M	758.3 s	996 ktps
LUBM8000	1068 M	1335 M	716.7 s	1863 ktps	1509 M	1824.8 s	827 ktps
DBPedia	394 M	401 M	409.9 s	1154 ktps	475 M	2886.6 s	165 ktps
CTD	335 M	358 M	70.2 s	5104 ktps	358 M	306.8 s	1176 ktps

- comparison for ρdf reasoning on LUBM:
  - WebPie (64 machines): 2125 kilo triples per second (ktps)
  - DynamiTE (single machine, stream reasoner): 227 ktps

Conclusion and future work

- we identified the memory consuming data structures of the RETE algorithm
  - restructured the working memories
  - introduced a compressed triple HashSet
  - were able to swap triples and working memories to the hard disk
- improved the reasoning performance by providing rule-specific code for the GPU

→ reasoning over **1 billion triples** on a laptop, throughput of up to **5 Mtps**

- future work may include:
  - adapt the proposed concepts to stream reasoning
  - adding more expressiveness

Fachhochschule
Dortmund
University of Applied Sciences and Arts

Thank you for your attention!

contact:

Martin Peters
martin.peters@fh-dortmund.de