Forward vs. Backward

Two approaches for web-scale reasoning

Jacopo Urbani <jacopo@cs.vu.nl>
Vrije Universiteit Amsterdam

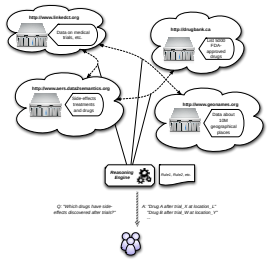
Reasoning on RDF

- Semantic Web: machines can "reason" on web data to infer new knowledge
- Reasoning can be performed by applying standardizes rules:

  if <a type B>
  and <B subclassOf C>
  then <a type C>

- RDFS ruleset: ~13 rules
- OWL pD*: ~23 rules
- OWL RL ruleset: ~78 rules

Reasoning on RDF

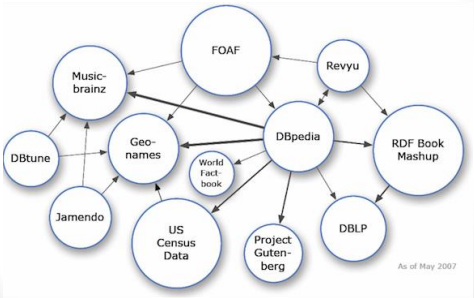


- This is not a problem. We know how to do this.

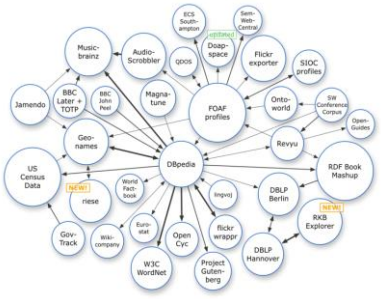
The problem of scale

- However, the Web is a bit larger than four websites…

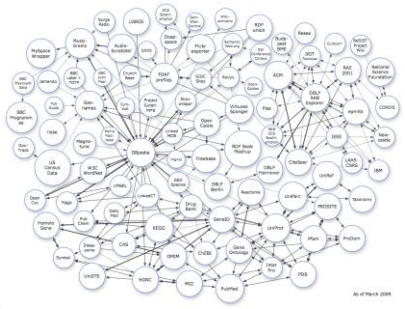
2007



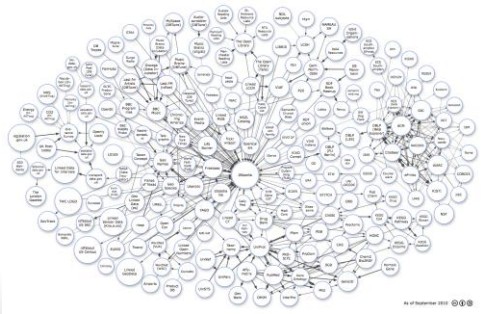
2008



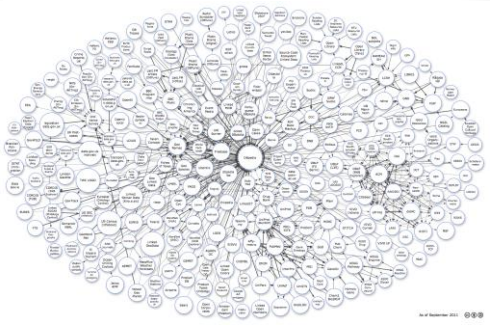
2009



2010



2011



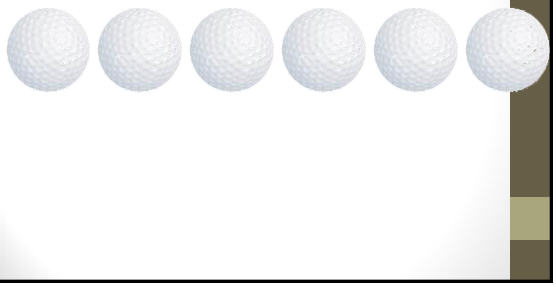
The problem of scale

- However, the Web is a bit larger than four websites…
  - Current LinkedDataCloud size (updated 2011): ~20 billion facts
  - Google Knowledge's Graph: 25 billions of formalized facts about billions of items. Volume is doubling every year
  - Expected outcome of OpenPhacts (large project with pharma companies): 20 - 50 billion triples, over 25 million mappings, over 20 data sources

1 Triple = 1 Golf ball

5 Triples

10 Triples

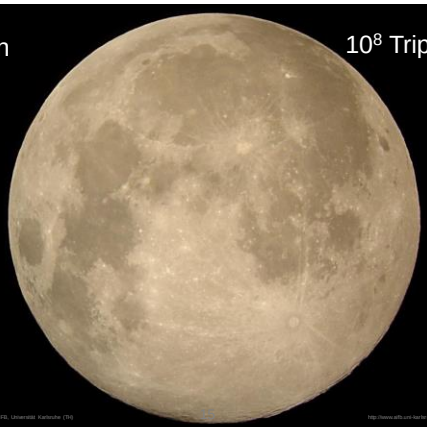


Suez Canal

10^7 Triples

Moon

10^8 Triples

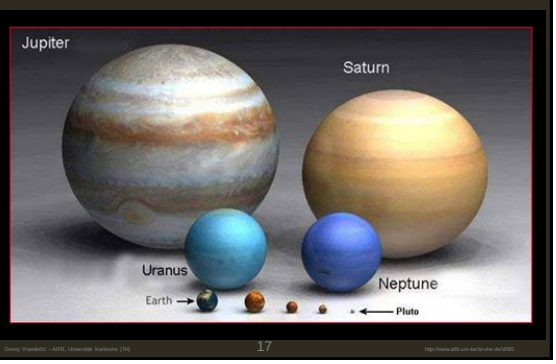


Earth

~10^9 Triples

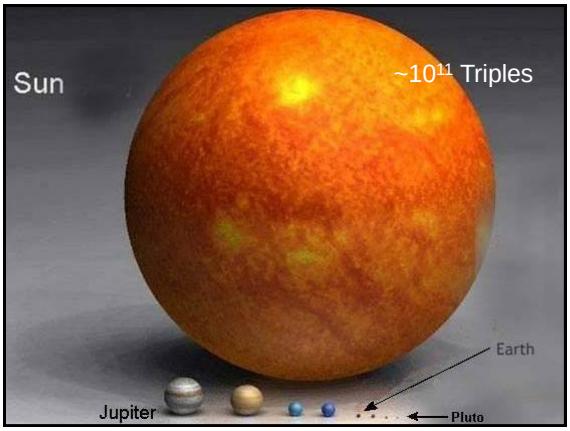
Jupiter

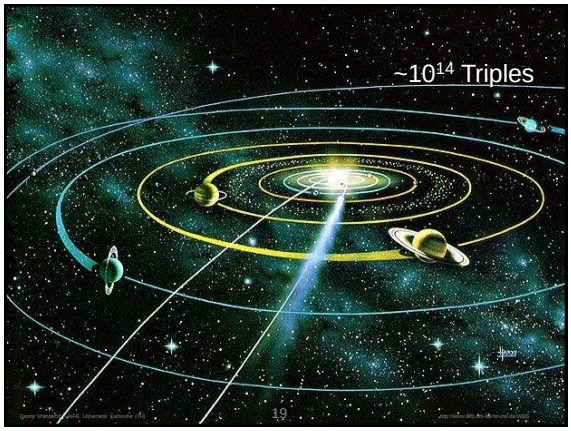
~10^{10} Triples ≈ 1 triple per web-page



Sun

~10^{11} Triples





- "How do we reason on such a scale?"

Problems and challenges

- One machine is not enough to store and process the Web

- We must distribute **data** and **computation**
- What architecture?
  - Several architectures of supercomputers
    - SIMD (single instruction/multiple data) processors, like graphic cards
    - Multiprocessing computers (many CPU shared memory)
    - Clusters (shared nothing architecture)
  - Algorithms depend on the architecture
- We focus on the implementation on clusters

Problems and challenges

- In a distributed environment the increase of performance comes at the price of new problems that we must face:
  - Load balancing
  - High I/O cost
  - Programming complexity

Problems and challenges: load balancing

- Cause: In many cases (like reasoning) some data is needed much more than other (e.g. schema triples)
- Effect: some nodes must work more to serve the others. This hurts scalability

Problems and challenges: high I/O cost

- Cause: data is distributed on several nodes and during reasoning the peers need to heavily exchange it
- Effect: hard drive or network speed become the performance bottleneck

Problems and challenges: programming complexity

- Cause: in a parallel setting there are many technical issues to handle
  - Fault tolerance
  - Data communication
  - Execution control
  - Etc.
- Effect: Programmers need to write much more code in order to execute an application on a distributed architecture

What we did

1. First we explored the possibility of performing forward-chaining reasoning. WebPIE
2. Then we researched for methods for doing backward-chaining reasoning. QueryPIE

What we did

1. First we explored the possibility of performing forward-chaining reasoning. WebPIE
2. Then we researched for methods for doing backward-chaining reasoning. QueryPIE

WebPIE

- **WebPIE** is a forward reasoner that uses MapReduce to execute the reasoning rules
- All code, documentation, tutorial etc. is available online.

http://cs.vu.nl/webpie/

- **WebPIE algorithm**:
  - **Input**: triples in N-Triples format
  - 1) Compress the data with dictionary encoding
  - 2) Launch reasoning
  - 3) Decompress derived triples
  - **Output**: triples in N-Triples format

1st step: compression

2nd step: reasoning

WebPIE: Overview

- RDFS reasoning
  - Set of 13 rules
  - All rules require at most one join between a "schema" triple and an "instance" triple
- OWL reasoning
  - Logic more complex => rules more difficult
  - The ter Horst fragment (pD*) provides a set of 23 new rules
  - Some rules require a join between instance triples
  - Some rules require multiple joins

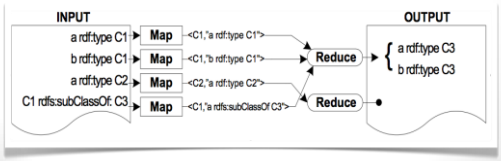
WebPIE: Overview

- **RDFS reasoning**
  - Set of 13 rules
  - All rules require at most one join between a "schema" triple and an "instance" triple
- OWL reasoning
  - Logic more complex => rules more difficult
  - The ter Horst fragment (pD*) provides a set of 23 new rules
  - Some rules require a join between instance triples
  - Some rules require multiple joins

WebPIE: RDFS reasoning

- How can we apply a reasoning rule with MapReduce?

if a rdf:type B
and B rdfs:subClassOf C
then a rdf:type C



WebPIE: RDFS reasoning

- Such straightforward way does not work
  - Load balancing
  - Duplicates derivation
  - etc.
- Three main optimizations
  1. Rules are applied in a specific order
  2. Schema triples are replicated in main memory
  3. Joins are performed in the *reduce* function and use the *map* function to generate less duplicates

WebPIE: Overview

- RDFS reasoning
  - Set of 13 rules
  - All rules require at most one join between a "schema" triple and an "instance" triple
- **OWL reasoning**
  - Logic more complex => rules more difficult
  - The ter Horst fragment (pD*) provides a set of 23 new rules
  - Some rules require a join between instance triples
  - Some rules require multiple joins

WebPIE: OWL reasoning

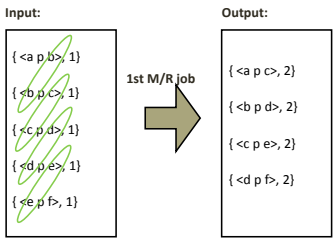
- RDFS optimizations are not enough. New ones are needed to deal with the more complex rules
- We will explain only one:
- Example:

if <p type TransitiveProperty>
and <a p b> and <b p c>
then <a p c>

- This rule is problematic because
  - Need to perform join between instance triples
  - Every time we derive also what we derived before
- Solution: we perform the join in the "naïve" way, but we only consider triples on a "specific" position

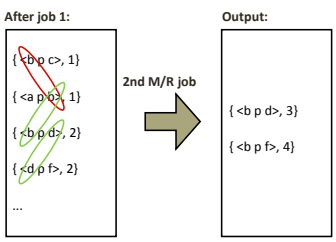
WebPIE: OWL reasoning

Example



WebPIE: OWL reasoning

Example

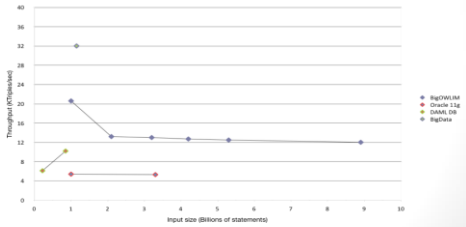


WebPIE: OWL reasoning

- By accepting only triples with a specific distance we avoid to derive information already derived
- General rule:
  - Every job accepts in input only triples derived in the previous two steps
  - During the execution of the *nth* job we derive only if:
    - The antecedent triples on the left side have distance 2^(n – 1) or 2^(n – 2)
    - The antecedent triples on the right side have distance greater than 2^(n – 2)

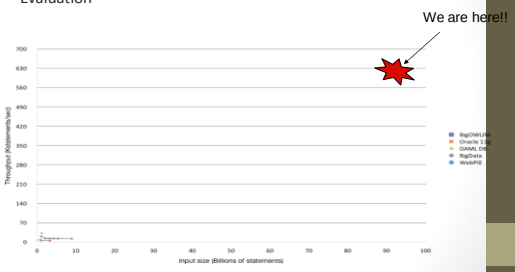
WebPIE: Forward-chaining

- Evaluation



WebPIE: Forward-chaining

- Evaluation



What we did

1. First we explored the possibility of performing forward-chaining reasoning. WebPIE
2. Then we researched for methods for doing backward-chaining reasoning. QueryPIE

Backward-chaining

- With WebPIE, we managed to perform reasoning over very large data
- However, the intrinsic problems of performing forward-chaining reasoning still remain
- Backward-chaining:
  - Advantage: we avoid to perform this (expensive) task
  - Disadvantage: we still need to perform reasoning at query-time

QueryPIE: Backward-chaining

- Reasoning is triggered only by SPARQL queries
- To reduce the runtime, *only* the schema is pre-materialized
  - does not change often
  - relatively small compared to the rest of the data
  - widely used in the reasoning rules
- QueryPIE is made of two algorithms
  - *pre-materialization algorithm:* executed before the user can query the knowledge base
  - *backward-chaining algorithm*: applied when the user launches a query

QueryPIE: Backward-chaining

Pre-materialization algorithm

- Goal: calculate all "schema" triples
- Problem: cannot use forward-chaining reasoning because instance-triples can generate schema
- Solution: We use the *backward-chaining algorithm:*
  - Problem: is incomplete if schema is not explicit
  - Solution: iterate the execution until we reached completeness

QueryPIE: Backward-chaining

Backward-chaining algorithm

- Inspired by QSQ algorithm (Datalog)
- Main features
  - 1) distributes the data across cluster nodes
  - 2) prunes the reasoning tree using the inferred schema
  - 3) replicates the schema and performs most locally
- Because of the pre-materialization, we can decrease the complexity significantly and improve the response time

QueryPIE: Backward-chaining

- Launched some example queries from the LUBM (10B triples) and LLD (5B triples)
- Cost pre-materialization:

Dataset	Reasoning time		N. iterations	N. derived triples
	Our approach	Full materialization		
LUBM	1s	4d4h16m	4	390
LLD	16m	5d10h45m	7	10 millions

QueryPIE: Backward-chaining

- Runtime of single pattern queries:

Query	Runtime (ms)		Processed Triples		I/O access	
	Cold	Warm	Total	Output	# lookups	MB
1	299.45	6.18	5	5	43	8
2	4767.17	131.6	463	239	12	205
3	186.06	5.65	3	1	18	5
4	310.53	10.5	37	29	86	8
5	382.21	13.9	1480	719	18	4
6	3986.91	2640.81	1599987	1599987	2	12
7	470.68	224.27	0	0	670	23
8	23.85	6.33	4466	4466	1	42
9	7705.91	616.42	140	128	3540	105
10	2609.52	1163.83	28446	26860	14372	337
11	2719.40	1914.28	8546	4504	15128	64
12	1954.97	2054.66	1187944	1187944	1	10

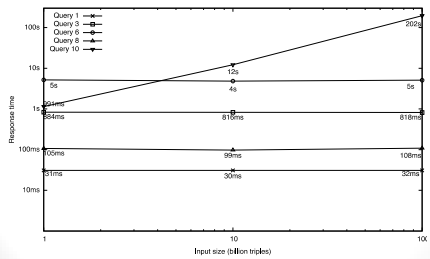
QueryPIE: Backward-chaining

- Runtime of the LUBM SPARQL queries:

Q.	Resp. time (ms.)		Results (#)
	Cold	Warm	
1	364.05	20.28	4
2	368.19	25.12	6
3	2041.20	493.89	34
4	1283.88	82.42	719
5	2061.27	361.11	4
6	11536.95	2784.72	7790
7	1745.20	367.96	4
8	1664.57	74.03	224
9	5717.32	1163.61	15
10	12304.29	886.68	37118

QueryPIE: Backward-chaining

- Response time if we increase the input size



What have we learnt?

We stepped out from the technical contribution to identify why it "works"…

Three important lessons:

- 1st lesson: Treat schema triples differently
- 2nd lesson: Data skew dominates the data distribution
- 3rd lesson: Certain problems appear only at a large scale

1st Lesson: treat the schema triples differently

- In the Semantic Web there is a big difference between "schema" triples and "instance" triples (A-Box vs. T-Box)
  - Schema triples: Few but very important for reasoning
  - Other triples: Many and not always used during reasoning
- Both in WebPIE and QueryPIE we replicated them on all the nodes and keep them in memory.
- Lesson: Identify the schema and treat it differently

2nd Lesson: Data skew dominates the data distribution

- Current web-data: high data skewness -> partitioning is crucial for performance
  - WebPIE: schema was replicated
  - QueryPIE: data was range-partitioned (good for small queries, bad for large queries)
- Lesson: Don't ignore data skewness. Otherwise your application will not scale to large numbers

3rd Lesson: Certain problems appear only at a large scale

- Web-scale reasoning requires high engineering efforts
  - WebPIE: ~15000 lines of code
  - QueryPIE: ~27000 lines of code
- Which programming language to use, libraries or compression techniques are crucial questions to reach certain performance.
- Lesson: Simple proof-of-concepts prototypes are not representative

Forward vs. Backward

- Both WebPIE and QueryPIE can scale to more than three times the size of the semantic web
- Problem: Both approaches are still fragile => with some bad data the derivation can explode
- Solution: *approximation* and *cost models*

"Reasoning on a web-scale is possible. We only started to discover it."