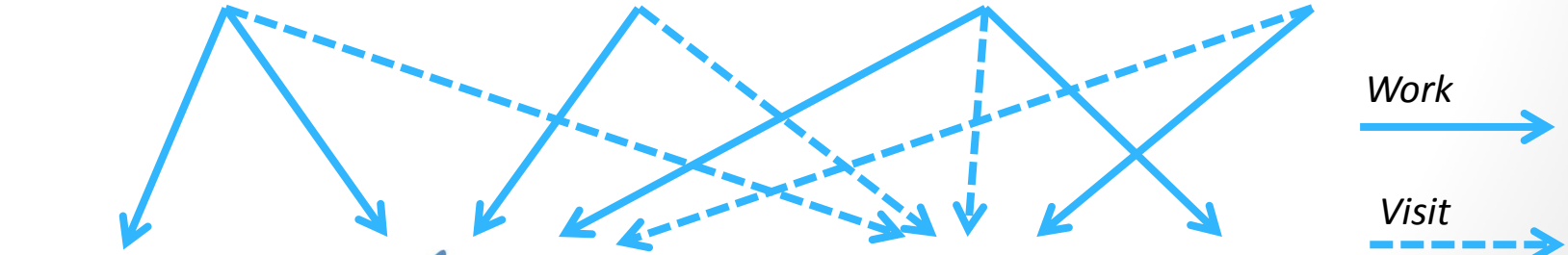
Robust Runtime Optimization and Skew-Resistant Execution of Analytical SPARQL Queries on Pig

Spyros Kotoulas, Jacopo Urbani, Peter Boncz, Peter Mika



IBM Research
vrije Universiteit amsterdam
YAHOO! RESEARCH
CWI
Centrum Wiskunde & Informatica

MapReduce

- MapReduce is a programming model originally developed by Google for large data processing
  - Conceptually very easy: first we apply a “map” function and then a “reduce” function
  - Parallelism and execution handled automatically by the framework (e.g. Hadoop)
  - Intensively used by Google, Yahoo!, Facebook, etc. for mainly for analytical tasks on large input (Web data) or other tasks IBM

Apache Pig

- To ease the programming with MapReduce Yahoo! Introduced Pig Latin (2008)

- Pig Latin is a SQL-like language that translates relational operators in Hadoop

- Example Pig Latin program:

```
IN = LOAD /path/ AS (S:chararray, P:chararray, O:chararray)
A = FILTER IN BY P eq “type”
B = FILTER IN BY P eq “foaf:friend”
C =JOIN A BY S, B BY S
DUMP C
```

Apache Pig

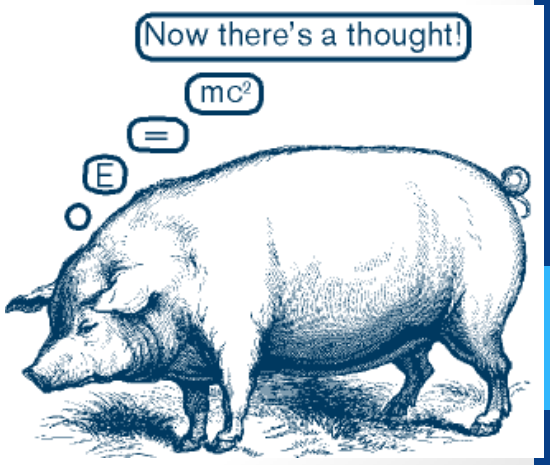
- Pig Latin does:
  - offer a simple script language that interfaces with any Hadoop cluster
  - group concurrent operations in single MapReduce jobs

- Pig Latin does not:
  - perform any optimization on the SPARQL query
  - perform any pre-analysis on the input to avoid load balancing

Existing work

- What the community has done:
  - Map SPARQL 1.0 to pig
  - Optimized for latency by pre-indexing data (e.g. loading in HBase)
  - Consolidate operations (joins) to reduce number of jobs

- What the community has not done:
  - Perform complex queries (SPARQL 1.1)
  - Sophisticated join order optimization



SPARQL on Pig

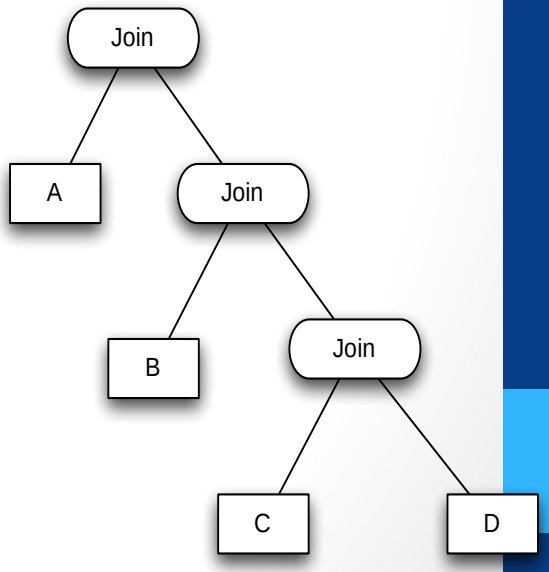
- We focus on the execution of very complex SPARQL queries that can be used in a ETL scenario using very large knowledge bases.
  - Pig has a very high startup cost that depends also on how busy the cluster is. Therefore, we cannot it to answer queries in an interactive usage
  - MapReduce is very inefficient in processing simple queries because it does not rely on indices. This means every time the input must be entirely read.

SPARQL on Pig

- Which **problems** did we address?
  1. The **cost model** for Pig is different than from traditional databases. Need to design appropriate query optimization technique.
  2. We have **no statistics** that could help us to choose the best execution strategy.
  3. We need to address the high **data skew** that is typical in RDF data with robust join algorithms.

SPARQL on Pig

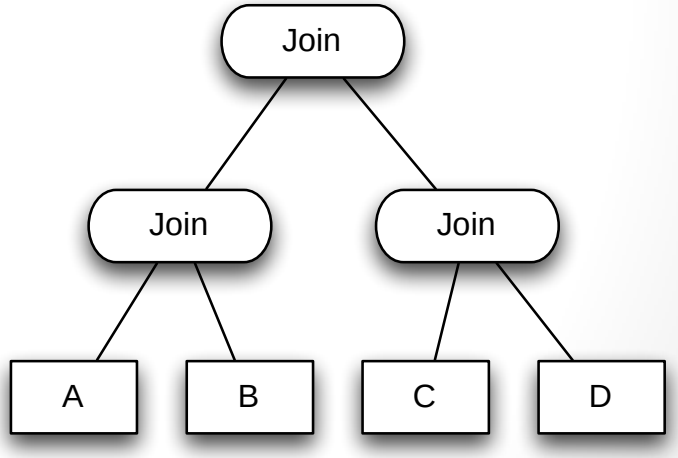
- **1st problem: query cost estimate for Pig**
- Our context is quite different from the typical database scenario
- Example: consider we want to join a, b, c ,d
  - There are different ways to perform this join
  - Databases rearrange the join order
    from the most selective to the least



SPARQL on Pig

- **1st problem: query cost estimate for Pig**

- In our case we might want to rearrange the tree differently
  - In Pig, operations are parallel
  - In evaluating the query plan, we privilege trees with shorter height to exploit the parallelization of operators, and reduce #jobs



SPARQL on Pig

- The 2nd and 3rd problems can only be addressed at run-time

- **2nd problem: no statistics in the input**
  - Databases have indexes on the data and keep statistics on cardinalities, join hit ratios etc
  - In our case we do not have any a-priori information on the data
  - One possible solution is to sample the join, however for complex joins you run out of samples very quickly

- **3rd problem: how do we perform the joins?**
  - In Pig we can execute a join in several way (during the map, during the reduce, etc.). We need to decide this in advance. What is the best solution under what circumstances?

SPARQL on Pig

- Our approach to the 2nd and 3rd problems: iterative sampling
  1. We select a set of the most promising joins and sample them
  2. We are **fully executing** the partial query plans leading up to the arguments of these joins
  3. We **sample** the promising joins and return to first point

- We use a cache to reuse results

Next: sampling and executing

SPARQL on Pig

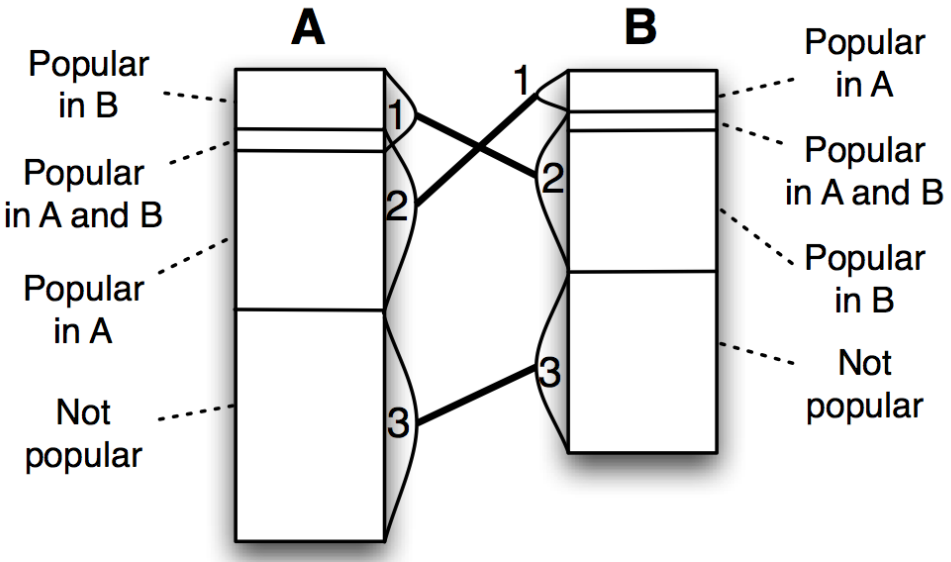
- **2nd problem: sampling the joins**
- Sampling a join to estimate its outcome is not trivial because data skew could influence the "real" computation
- To provide for an accurate estimation, we have implemented bifocal sampling in Pig

1: **AAAAAAA**AAAAAAB**A**B**AA**
2: **ACFDVDXFR**ESDFRFCDDX

10 hits

1: **AAAAAAA**AAAAAAB**A**B**AA**
2: ACFDVDXFR**ESDFRFCDDX**

0 hits



SPARQL on Pig

- **3rd problem: handle load balancing issues during the joins**

- In Pig, we can perform a join in two ways:
  - **Standard**: the join is performed on the reduce phace
  - **Replicated**: One side is loaded in memory and the join is performed on the map phase *(faster)*
- The classic join suffers from load-balancing problems if data has high skew
- The replicated join does not work if both sides are large.

SPARQL on Pig

- **3rd problem: handle load balancing issues**
- Current approach: we combine both joins. Popular side join is performed on the map phase, the other in the reduce
- *Operations:*
  - *1) sample the cardinality of the join and split one side between the triples which cause load balancing issues and the others*
  - *2) perform a replicated join only using first set (the popular one)*
  - *3) perform a classic join against the second set*

Evaluation (on BSBM BI)

DAS4 – 32 dual core/4G RAM
Y! – 3500 8-core/16G RAM, shared

Loading time: 0

Our Approach

Query	1B DAS4	1B Y!	10B Y!
1	38m	1h50m	1h17m
2	13m	23m	31m
3	17m	18m	24m
4	38m	1h34m	54m
5	1h4m	3h10m	1h52m
6	48m	34m	1h19m
7	26m	43m	46m
8	1h	1h59m	1h38m

Sum: **8 hours**

Single server – 40 Cores/1TB RAM

Loading time: 61 hours

Virtuoso v7

Query	Cold runtime	Warm runtime
1	3m46s	1m48s
2	41s	15s
3	29m	24m5s
4	52m6s	50m55s
5	11m42s	5m19s
6	6s	0.06s
7	50s	9ms
8	39m58s	36m22s

Sum: **2 hours**

Evaluation (on Web crawl)

26 Billion triple Web Crawl

Runtime

```
SELECT (count(?s) as ?f) (min(?s) as ?ex) ?ct ?di ?mx ?mi{
      {SELECT ?s (count(?s) as ?ct) (count(distinct ?p) as ?di) (max(?p) as ?mx)
            (min(?p) as ?mi) {?s ?p ?o.} GROUP BY ?s}
      GROUP BY ?ct ?di ?mx ?mi ORDER BY desc(?f) LIMIT 10000}
```

1h 29m

```
SELECT ?p (COUNT(?s) AS ?c) {?s ?p ?o.} GROUP BY ?p ORDER BY ?c
```

29m

```
SELECT ?C (COUNT(?s) AS ?n ) {?s a ?C.} GROUP BY ?C ORDER BY ?n
```

25m

Conclusions

- We have developed an approach to execute SPARQL 1.1 queries
  - Pig-aware planning
  - Dynamic query optimization
  - Robust and accurate sampling
  - Skew-resistant joins
- We have verified that it is suitable for complex analysis tasks/ETL like queries.
- Compared with Virtuoso running on large server. Conclusions are that our approach is better for posting few expensive queries, worse otherwise.

Future

- We have **not** taken into account some techniques from the literature
  - E.g. make custom functions to group joins in fewer jobs
- Reduce the number of jobs by indexing part of the input
  - Trade-off between loading and execution time
- Implement rule-based reasoning based in this technology
  - Our skew-resistant joins would be critical in this respect

END

Questions?

Acknowledgements:

Financially supported by EU FP7 LarKC & Yahoo Faculty Engagement Programme