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

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.
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 2\textsuperscript{nd} and 3\textsuperscript{rd} problems can only be addressed at run-time

2\textsuperscript{nd} 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

3\textsuperscript{rd} 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 2\textsuperscript{nd} and 3\textsuperscript{rd} problems: \textit{iterative sampling}
  1. We select a set of the most promising joins and sample them
  2. We are \textbf{fully executing} the partial query plans leading up to the arguments of these joins
  3. We \textbf{sample} the promising joins and return to first point

• We use a cache to reuse results

\textit{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
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 phase
  • **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)

**Our Approach**

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

**Loading time:** 0

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

**Sum:** 8 hours

**Virtuoso v7**

Single server – 40 Cores/1TB RAM  

**Loading time:** 61 hours

<table>
<thead>
<tr>
<th>Query</th>
<th>Cold runtime</th>
<th>Warm runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3m46s</td>
<td>1m48s</td>
</tr>
<tr>
<td>2</td>
<td>41s</td>
<td>15s</td>
</tr>
<tr>
<td>3</td>
<td>29m</td>
<td>24m5s</td>
</tr>
<tr>
<td>4</td>
<td>52m6s</td>
<td>50m55s</td>
</tr>
<tr>
<td>5</td>
<td>11m42s</td>
<td>5m19s</td>
</tr>
<tr>
<td>6</td>
<td>6s</td>
<td>0.06s</td>
</tr>
<tr>
<td>7</td>
<td>50s</td>
<td>9ms</td>
</tr>
<tr>
<td>8</td>
<td>39m58s</td>
<td>36m22s</td>
</tr>
</tbody>
</table>

**Sum:** 2 hours
Evaluation (on Web crawl)

26 Billion triple Web Crawl

```
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}

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

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

Acknowledgements:
Financially supported by EU FP7 LarKC & Yahoo Faculty Engagement Programme