Cascading Map–Side Joins over HBase for Scalable Join Processing

Joint Workshop on Scalable and High-Performance Semantic Web Systems (SSWS + HPCSW 2012)
Collocated with the 11th International Semantic Web Conference (ISWC 2012)

Alexander Schätzle
Martin Przyjaciel-Zablocki
Christopher Dorner
Thomas Hornung
Georg Lausen

University of Freiburg
Databases & Information Systems
Motivation

- RDF datasets are growing constantly (e.g. LOD)
- Querying RDF datasets at web-scale is challenging

Our Approach
- Distributed scalable RDF engine for processing very large datasets (RDF + SPARQL)
  - Build on common & widely-used frameworks (Hadoop MapReduce, HBase, Pig, Cassandra, …)
Previous Work – PigSPARQL [1]

- **SPARQL on top of Pig Latin**

- **Advantages**
  - All operators of SPARQL 1.0
  - Benefits from Pig optimizations
  - Runs "out-of-the-box" on Hadoop
  - Portable on other platforms

- **Performance**
  - Good scalability and performance for complex analytical queries
  - Performance not satisfying for more selective queries

- **Reasons**
  - Reduce-Side Join (→ Data shuffling)
  - No built-in index structures

New Approach

- Store input dataset in HBase instead of plain HDFS
- Process the join in the Map phase to avoid unnecessary data shuffling

**Expected benefit**
- No costly Shuffle & Sort phase
- I/O reduction due to HBase indexes

**Expected drawbacks**
- Communication overhead
- Significantly higher RAM consumption
- Not ideal for high-output queries
RDF Storage in HBase

Store RDF in a NoSQL data store
What is HBase (Not)?

- **Clone of Google's Bigtable**
  - Column-oriented, semi-structured NoSQL data store
  - Distributed over many machines (Hadoop Cluster)
  - Layered on top of HDFS (Hadoop Distributed File System)
    - Files split into blocks (e.g. 64MB) and replicated across machines
    - Tolerant of machine failure
  - Adds **random data access** to HDFS in "close to real-time"
  - Strictly consistent!

- **Not a relational query engine**
  - Not designed for normalized schemas
  - No join operators
  - No expressive query language like SQL
HBase Data Model

- **Sparse, distributed, sorted, multidimensional map**
  - Indexed by row key
  - Values can have multiple versions, identified via timestamps
  - Columns are grouped into column families
  - Tables are dynamically split into regions
  - Every region is assigned to exactly one Region Server

- **Access Pattern:**
  \[(Table, RowKey, Family, Column, Timestamp) \rightarrow Value\]
RDF Storage by Example (1)

"PigSPARQL" \(\xrightarrow{\text{title}}\) Article1

"2011" \(\xrightarrow{\text{year}}\) Alex

author

cite

Martin \(\xleftarrow{\text{author}}\) Article2

author

year

"2011"

p:title \(\xrightarrow{}\) "PigSPARQL"
p:year \(\xrightarrow{}\) "2011"
p:author \(\xrightarrow{}\) {Alex, Martin}

Article1

p:title \(\xrightarrow{}\) "RDFPath"
p:year \(\xrightarrow{}\) "2011"
p:author \(\xrightarrow{}\) {Martin, Alex}
p:cite \(\xrightarrow{}\) {Article1}

Article2

---

\(T_{s-po}\):

<table>
<thead>
<tr>
<th>rowkey</th>
<th>family:column (\xrightarrow{}) value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Article1</td>
<td>p:title (\xrightarrow{}) {&quot;PigSPARQL&quot;}, p:year (\xrightarrow{}) {&quot;2011&quot;}, p:author (\xrightarrow{}) {Alex, Martin}</td>
</tr>
<tr>
<td>Article2</td>
<td>p:title (\xrightarrow{}) {&quot;RDFPath&quot;}, p:year (\xrightarrow{}) {&quot;2011&quot;}, p:author (\xrightarrow{}) {Martin, Alex}, p:cite (\xrightarrow{}) {Article1}</td>
</tr>
</tbody>
</table>

\(T_{o-ps}\):

<table>
<thead>
<tr>
<th>rowkey</th>
<th>family:column (\xrightarrow{}) value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alex</td>
<td>p:author (\xrightarrow{}) {Article1, Article2}</td>
</tr>
<tr>
<td>Article1</td>
<td>p:cite (\xrightarrow{}) {Article2}</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

---

Cascading Map-Side Joins over HBase for Scalable Join Processing
## Triple Pattern Matching

<table>
<thead>
<tr>
<th>pattern</th>
<th>table</th>
<th>filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s, p, o)</td>
<td>$T_{s_po}$ or $T_{o_ps}$</td>
<td>column &amp; value</td>
</tr>
<tr>
<td>(?s, p, o)</td>
<td>$T_{o_ps}$</td>
<td>column</td>
</tr>
<tr>
<td>(s, ?p, o)</td>
<td>$T_{s_po}$ or $T_{o_ps}$</td>
<td>value</td>
</tr>
<tr>
<td>(s, p, ?o)</td>
<td>$T_{s_po}$</td>
<td>column</td>
</tr>
<tr>
<td>(?s, ?p, o)</td>
<td>$T_{o_ps}$</td>
<td></td>
</tr>
<tr>
<td>(?s, p, ?o)</td>
<td>$T_{s_po}$ or $T_{o_ps}$ (SCAN)</td>
<td>column</td>
</tr>
<tr>
<td>(s, ?p, ?o)</td>
<td>$T_{s_po}$</td>
<td></td>
</tr>
<tr>
<td>(?s, ?p, ?o)</td>
<td>$T_{s_po}$ or $T_{o_ps}$ (SCAN)</td>
<td></td>
</tr>
</tbody>
</table>

**server side filters**
MAPSIN Join

Map–Side Index Nested Loop Join
Cascading Map-Side Joins over HBase for Scalable Join Processing
# Multiway Join Optimization

<table>
<thead>
<tr>
<th>Query pattern</th>
<th>Corresponding HBase requests</th>
</tr>
</thead>
</table>
| ?article title ?title | 1. iteration  
(T<sub>s_po</sub>, article1, column=author)  
(T<sub>s_po</sub>, article2, column=author) |
| ?article author ?author | 2. iteration  
(T<sub>s_po</sub>, article1, column=year)  
(T<sub>s_po</sub>, article2, column=year) |
| ?article year ?year | rowkey, filter |

Query pattern

- `?article title ?title`
- `?article author ?author`
- `?article year ?year`

Corresponding HBase requests

1. iteration
- (T<sub>s_po</sub>, article1, column=author)
- (T<sub>s_po</sub>, article2, column=author)

2. iteration
- (T<sub>s_po</sub>, article1, column=year)
- (T<sub>s_po</sub>, article2, column=year)

Rowkey and Filter
## Multiway Join Optimization

<table>
<thead>
<tr>
<th>Query pattern</th>
<th>Corresponding HBase requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>?article title ?title</td>
<td>1. iteration (Ts_po, article1, column=author)</td>
</tr>
<tr>
<td></td>
<td>(Ts_po, article2, column=author)</td>
</tr>
<tr>
<td>?article author ?author</td>
<td>2. iteration (Ts_po, article1, column=year)</td>
</tr>
<tr>
<td></td>
<td>(Ts_po, article2, column=year)</td>
</tr>
</tbody>
</table>

### Cascading Map-Side Joins over HBase for Scalable Join Processing
Multiway Join Optimization

<table>
<thead>
<tr>
<th>Query pattern</th>
<th>Corresponding HBase requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>?article title ?title</td>
<td>(T_{s,po}, article1, column=author)</td>
</tr>
<tr>
<td></td>
<td>(T_{s,po}, article2, column=author)</td>
</tr>
<tr>
<td>?article author ?author</td>
<td>(T_{s,po}, article1, column=year)</td>
</tr>
<tr>
<td></td>
<td>(T_{s,po}, article2, column=year)</td>
</tr>
<tr>
<td>?article year ?year</td>
<td>(T_{s,po}, article1, column=author)</td>
</tr>
<tr>
<td></td>
<td>(T_{s,po}, article1, column=year)</td>
</tr>
<tr>
<td></td>
<td>(T_{s,po}, article2, column=author)</td>
</tr>
<tr>
<td></td>
<td>(T_{s,po}, article2, column=year)</td>
</tr>
</tbody>
</table>
## Multiway Join Optimization

<table>
<thead>
<tr>
<th>Query pattern</th>
<th>Corresponding HBase requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>?article title ?title</td>
<td>$(T_{s,po}, \text{article1}, \text{column}\text{=author})$, $(T_{s,po}, \text{article2}, \text{column}\text{=author})$</td>
</tr>
<tr>
<td>?article author ?author</td>
<td>1. iteration</td>
</tr>
<tr>
<td>?article year ?year</td>
<td>2. iteration</td>
</tr>
<tr>
<td>?article title ?title</td>
<td>(1. iteration)</td>
</tr>
<tr>
<td>?article author ?author</td>
<td>(1. iteration)</td>
</tr>
<tr>
<td>?article year ?year</td>
<td>(1. iteration)</td>
</tr>
<tr>
<td></td>
<td>(4 requests!)</td>
</tr>
</tbody>
</table>

---

Cascading Map-Side Joins over HBase for Scalable Join Processing
Multiway Join Optimization

<table>
<thead>
<tr>
<th>Query pattern</th>
<th>Corresponding HBase requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>?article title ?title</td>
<td>1. iteration</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>?article author ?author</td>
<td>2. iteration</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>?article year ?year</td>
<td>1. iteration</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. iteration

2. iteration

2 requests!
Evaluation

Lehigh University Benchmark (LUBM)
Evaluation Setup

- **Cluster Hardware**
  - 10 Dell PowerEdge R200 servers
  - Dual Core 3.16 GHz CPU
  - 8 GB RAM
  - 3 TB hard disk
  - Gigabit Network

- **Frameworks**
  - Hadoop 0.20.2 (CDH3)
  - HBase 0.90.4

- **Datasets**
  - 1000 – 3000 LUBM universities
  - ~ 210 – 630 million triples (after reasoning)
LUBM Q1

- Base Case (single join)
- Linear Scaling behavior for both approaches
- MAPSIN performs 8 – 13 times faster than PigSPARQL

```
SELECT ?X
WHERE {
  ?X rdf:type ub:GraduateStudent .
  ?X ub:takesCourse <...GraduateCourse0>
}
```
General Case (sequence of joins), Multiway Join Optimization applicable

- Linear Scaling behavior for both approaches
- MAPSIN performs up to 28 times faster than PigSPARQL
- MAPSIN multiway join ~ 3 times faster than standard MAPSIN

SELECT ?X ?Y1 ?Y2 ?Y3
WHERE {
  ?X rdf:type ub:Professor .
  ?X ub:worksFor <...Department0.University0.edu> .
  ?X ub:telephone ?Y3
}
Conclusion & Future Work

**Conclusion**
- MAPSIN joins are processed completely in Map phase
- MAPSIN joins are easily iterable in a sequence of joins (without auxiliary Shuffle & Reduce Phases)
- Multiway join optimization reduces the number of iterations and HBase requests
- Outperforms reduce-side joins (PigSPARQL) by an order of magnitude (depending on the query selectivity)
- Performance degrades with increasing query output

**Future Work**
- Improvements of the RDF storage schema
- Incorporate MAPSIN joins into PigSPARQL

[http://www.superscholar.org]
Thank you for your attention!
Backup Slides

Things not mentioned yet
Evaluation Runtimes

- Runtimes for PigSPARQL (P) and MAPSIN (M)

<table>
<thead>
<tr>
<th>LUBM</th>
<th>1000</th>
<th></th>
<th>1500</th>
<th></th>
<th>2000</th>
<th></th>
<th>2500</th>
<th></th>
<th>3000</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>M</td>
<td>P</td>
<td>M</td>
<td>P</td>
<td>M</td>
<td>P</td>
<td>M</td>
<td>P</td>
<td>M</td>
</tr>
<tr>
<td>Q1</td>
<td>324</td>
<td>34</td>
<td>475</td>
<td>51</td>
<td>634</td>
<td>53</td>
<td>790</td>
<td>70</td>
<td>944</td>
<td>84</td>
</tr>
<tr>
<td>Q3</td>
<td>324</td>
<td>33</td>
<td>480</td>
<td>42</td>
<td>642</td>
<td>49</td>
<td>805</td>
<td>59</td>
<td>961</td>
<td>72</td>
</tr>
<tr>
<td>Q4</td>
<td>1202</td>
<td>121</td>
<td>1758</td>
<td>167</td>
<td>2368</td>
<td>182</td>
<td>2919</td>
<td>235</td>
<td>3496</td>
<td>279</td>
</tr>
<tr>
<td>Q4 MJ</td>
<td>861</td>
<td>37</td>
<td>1297</td>
<td>53</td>
<td>1728</td>
<td>62</td>
<td>2173</td>
<td>81</td>
<td>2613</td>
<td>92</td>
</tr>
<tr>
<td>Q5</td>
<td>329</td>
<td>33</td>
<td>484</td>
<td>44</td>
<td>640</td>
<td>53</td>
<td>800</td>
<td>66</td>
<td>955</td>
<td>80</td>
</tr>
<tr>
<td>Q6</td>
<td>149</td>
<td>48</td>
<td>214</td>
<td>60</td>
<td>284</td>
<td>69</td>
<td>355</td>
<td>84</td>
<td>424</td>
<td>104</td>
</tr>
<tr>
<td>Q7</td>
<td>1013</td>
<td>62</td>
<td>1480</td>
<td>68</td>
<td>1985</td>
<td>93</td>
<td>2472</td>
<td>114</td>
<td>2928</td>
<td>123</td>
</tr>
<tr>
<td>Q8</td>
<td>1172</td>
<td>64</td>
<td>1731</td>
<td>77</td>
<td>2318</td>
<td>33</td>
<td>2870</td>
<td>108</td>
<td>3431</td>
<td>121</td>
</tr>
<tr>
<td>Q11</td>
<td>319</td>
<td>33</td>
<td>469</td>
<td>46</td>
<td>620</td>
<td>53</td>
<td>780</td>
<td>69</td>
<td>931</td>
<td>79</td>
</tr>
<tr>
<td>Q13</td>
<td>325</td>
<td>44</td>
<td>482</td>
<td>72</td>
<td>645</td>
<td>84</td>
<td>800</td>
<td>108</td>
<td>957</td>
<td>128</td>
</tr>
<tr>
<td>Q14</td>
<td>149</td>
<td>43</td>
<td>214</td>
<td>70</td>
<td>288</td>
<td>79</td>
<td>364</td>
<td>89</td>
<td>434</td>
<td>107</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SP²Bench</th>
<th>200M</th>
<th></th>
<th>400M</th>
<th></th>
<th>600M</th>
<th></th>
<th>800M</th>
<th></th>
<th>1000M</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>M</td>
<td>P</td>
<td>M</td>
<td>P</td>
<td>M</td>
<td>P</td>
<td>M</td>
<td>P</td>
<td>M</td>
</tr>
<tr>
<td>Q1</td>
<td>545</td>
<td>58</td>
<td>1026</td>
<td>118</td>
<td>1527</td>
<td>153</td>
<td>2018</td>
<td>177</td>
<td>2519</td>
<td>214</td>
</tr>
<tr>
<td>Q1 MJ</td>
<td>310</td>
<td>42</td>
<td>600</td>
<td>87</td>
<td>896</td>
<td>118</td>
<td>1187</td>
<td>154</td>
<td>1476</td>
<td>174</td>
</tr>
<tr>
<td>Q2 MJ</td>
<td>1168</td>
<td>241</td>
<td>2341</td>
<td>444</td>
<td>3514</td>
<td>671</td>
<td>4745</td>
<td>834</td>
<td>6005</td>
<td>999</td>
</tr>
<tr>
<td>Q3a</td>
<td>227</td>
<td>70</td>
<td>435</td>
<td>139</td>
<td>641</td>
<td>178</td>
<td>845</td>
<td>235</td>
<td>1050</td>
<td>274</td>
</tr>
<tr>
<td>Q10</td>
<td>99</td>
<td>40</td>
<td>174</td>
<td>84</td>
<td>254</td>
<td>111</td>
<td>340</td>
<td>151</td>
<td>414</td>
<td>167</td>
</tr>
</tbody>
</table>
MapReduce

- **Automatic parallelization of computations**

- **Distributed File System**
  - Commodity hardware → Fault tolerance by replication
  - Very large files / write-once, read-many pattern

- **Apache Hadoop**
  - Well-known open-source implementation
Map–Side Joins in MapReduce

- Map–Side (Merge) Join
  - Input datasets must be:
    1. divided into same number of partitions
    2. Sorted by the same key (the join key)
    3. All records of a particular key must reside in the same partition
  - Problem: Fulfill requirements for subsequent iterations

- Broadcast Join
  - One dataset small enough to be distributed to each node
  - Problem: Not feasible for big datasets