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, …)
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
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
  - 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" | title | Article1
--- | --- | ---
"2011" | year | Article1

Alex | author | Article1

Martin | author | Article2

"2011" | year | Article2

"RDFPath" | title | Article2

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

Ts_po:

To_ps:

# 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
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
MAPSIN Join

1. SCAN for local mappings: `?article title ?title`

2. map inputs
   - `?article=article1 ?title="PigSPARQL"`
   - `?article=article2 ?title="RDFPath"`

3. GET bindings: `article1 author ?author`
   - `?article=article1 ?title="PigSPARQL" ?author=Alex`
   - `?article=article1 ?title="PigSPARQL" ?author=Martin`
   - `?article=article2 ?title="RDFPath" ?author=Martin`
   - `?article=article2 ?title="RDFPath" ?author=Alex`

4. map outputs
   - `?article=article1 ?title="PigSPARQL" ?author=Martin`
   - `?article=article1 ?title="PigSPARQL" ?author=Alex`
   - `?article=article2 ?title="RDFPath" ?author=Martin`
   - `?article=article2 ?title="RDFPath" ?author=Alex`

Cascading Map-Side Joins over HBase for Scalable Join Processing

```sparql
SELECT *
WHERE {
  ?article title ?title .
  ?article author ?author .
  ?article year ?year
}
```
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>?article author ?author</td>
<td>2. iteration</td>
</tr>
<tr>
<td>?article year ?year</td>
<td></td>
</tr>
</tbody>
</table>

1. iteration

(\(T_{s,po}\), article1, column=author)
(\(T_{s,po}\), article2, column=author)

2. iteration

(\(T_{s,po}\), article1, column=year)
(\(T_{s,po}\), article2, column=year)

rowkey  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></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>?article author ?author</td>
<td>2. iteration (Ts_po, article1, column=year)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>?article year ?year</td>
<td></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>(Ts_po, article1, column=author) (Ts_po, article2, column=author)</td>
</tr>
<tr>
<td>?article author ?author</td>
<td></td>
</tr>
<tr>
<td>?article year ?year</td>
<td>(Ts_po, article1, column=year) (Ts_po, article2, column=year)</td>
</tr>
</tbody>
</table>

1. iteration

2. iteration

---

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  
(Ts_po, article1, column=author)  
(Ts_po, article2, column=author) |
| ?article author ?author |                          |
| ?article year ?year | 2. iteration  
(Ts_po, article1, column=year)  
(Ts_po, article2, column=year) |

For another query pattern:

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

This results in 4 requests!
### Multiway Join Optimization

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

Cascading Map-Side Joins over HBase for Scalable Join Processing
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

```sql
SELECT ?X
WHERE {
  ?X rdf:type ub:GraduateStudent .
  ?X ub:takesCourse <...GraduateCourse0>
}
```
LUBM Q4

- 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!