Rya: A Scalable RDF Triple Store for the Clouds

R. Punnose¹, Adina Crăiniceanu², D. Rapp³

¹-Proteus Technologies, ²-U.S. Naval Academy, ³-LTS

Presentation at Cloud Intelligence Workshop 2012, a VLDB 2012 workshop
RDF Data

- Increasingly popular
- Based on making statements about resources
  - Statements are formed as triples (subject-predicate-object)
  - Example, “The sky has the color blue”
    - Subject = The sky
    - Predicate = has color
    - Object = blue
Why RDF?

- W3C standard
- Large community/tool support
- Easy to understand
- Intrinsically represents a labeled, directed graph

- Unstructured
  - Though with RDFS/OWL, can add structure
Why Not RDF?

- **Storage**
  - Stores can be large for small amounts of data
- **Speed**
  - Slow to answer simple questions
- **Scale**
  - Not easy to scale with size of data
Rya – Distributed RDF Triple Store

- Smartly store RDF data in Accumulo
  - Scalability
  - Load balance
- Build on the OpenRDF interface implementation for SPARQL
  - Fast queries
Outline

- Problem
- Background
- Rya
  - Triple index
  - Performance enhancements
- Experimental results
- Conclusions and future work
OpenRDF Sesame

- Utilities to parse, store, and query RDF data
- Supports SPARQL
- Ex: SELECT ?x WHERE {
  ?x rdf:type Faculty .
  ?x degreeFrom Cornell .}
- SPARQL queries evaluated based on triple patterns
  - Ex: (*, rdf:type, Faculty)
Accumulo

- Google BigTable implementation

- Compressed, Distributed, Scalable
- Adds security, row level authentication/visibility, etc
- The Accumulo store acts as persistence and query backend to OpenRDF
Outline

- Problem
- Background
- Rya
  - Triple index
  - Performance enhancements
- Experimental results
- Conclusions and future work
Architectural Overview - Rya

Query Processing

Inference
Query Optimization Engine
Query Execution

Storage And Inference Layer Implementation

Data Storage

Rya ********
Triple Table Index

- 3 Tables
  - SPO: subject, predicate, object
  - POS: predicate, object, subject
  - OSP: object, subject, predicate
- Store triples in the RowID of the table
- Take advantage of lexicographical sorting of row keys → fast range queries
- All patterns can be translated into a scan of one of these tables
Sample Triple Storage

Example RDF triple:

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>degreeFrom</td>
<td>Cornell</td>
</tr>
</tbody>
</table>

Stored RDF triple in Accumulo tables:

<table>
<thead>
<tr>
<th>Table</th>
<th>Stored Triple</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPO</td>
<td>Alice, degreeFrom, Cornell</td>
</tr>
<tr>
<td>POS</td>
<td>degreeFrom, Cornell, Alice</td>
</tr>
<tr>
<td>OSP</td>
<td>Cornell, Alice, degreeFrom</td>
</tr>
<tr>
<td>Triple Pattern</td>
<td>Table to Scan</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>(Alice, degreeFrom, Cornell)</td>
<td>Any table (SPO default)</td>
</tr>
<tr>
<td>(Alice, degreeFrom, *)</td>
<td>SPO</td>
</tr>
<tr>
<td>(Alice, *, Cornell)</td>
<td>OSP</td>
</tr>
<tr>
<td>(*, degreeFrom, Cornell)</td>
<td>POS</td>
</tr>
<tr>
<td>(Alice, *, *)</td>
<td>SPO</td>
</tr>
<tr>
<td>(*, degreeFrom, *)</td>
<td>POS</td>
</tr>
<tr>
<td>(*, *, Cornell)</td>
<td>OSP</td>
</tr>
<tr>
<td>(*, *, *)</td>
<td>any full table scan (SPO default)</td>
</tr>
</tbody>
</table>
SELECT ?x WHERE {
  ?x takesCourse DBCourse .
  ?x rdf:type GraduateStudent .
}

Step 1: POS – scan range

| rdf:type, Professor, Alice |
| takesCourse, AI Course, John |
| takesCourse, AI Course, Zack |
| takesCourse, DBCourse, Bob |
| takesCourse, DBCourse, Greta |
| takesCourse, DBCourse, John |
| takesCourse, HCI Course, Alice |

Step 2: for each ?x, SPO – index lookup

| Bob, rdf:type, UndergradStudent |
| Greta, rdf:type, GraduateStudent |
| John, rdf:type, GraduateStudent |
Query Processing using Inference

```
SELECT ?x WHERE { ?x rdf:type Faculty  }
```

New query: SELECT ?x WHERE {
  ?type rdfs:subClassOf Faculty .
  ?x rdf:type ?type }

Rya ********
Query Plan for Expanded Query

SELECT ?x WHERE {
    ?type rdfs:subClassOf Faculty.
    ?x rdf:type ?type . }

Step 1: POS – scan range

| ... |
| ... |
| ... |
| ... |
| rdf:subClassOf, Faculty, AssistProf |
| rdf:subClassOf, Faculty, AssocProf |
| rdf:subClassOf, Faculty, Professor |
| ... |
| ... |

Step 2: For each ?type, POS – scan range

| ... |
| rdf:type, AssocProf, Amelia |
| rdf:type, AssocProf, George |
| rdf:type, Professor, Alice |
| ... |
Performance Enhancements

- Statistics Collection
- Parallel Joins
- Accumulo Batch Scanner use
  - Decreases network connections by up to 1K fold
- Time Ranges
  - Allow RDF querying on a small subset of data (based on a time loaded)
Optimized Joins with Statistics

- Collect statistics about data distribution
- Most selective triple evaluated first
- Ex:

<table>
<thead>
<tr>
<th>Value</th>
<th>Role</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdf:type</td>
<td>Predicate</td>
<td>1mil</td>
</tr>
<tr>
<td>Student</td>
<td>Object</td>
<td>400K</td>
</tr>
<tr>
<td>takesCourse</td>
<td>Predicate</td>
<td>800K</td>
</tr>
<tr>
<td>DBCourse</td>
<td>Object</td>
<td>200</td>
</tr>
</tbody>
</table>

SELECT ?x WHERE { ?x takesCourse DBCourse . ?x rdf:type Student . } Vs. SELECT ?x WHERE { ?x rdf:type Student . ?x takesCourse DBCourse }
Parallel Joins

SELECT ?x WHERE {
  ?type rdfs:subClassOf Faculty.
  ?x rdf:type ?type .
}  

Step 1: POS – scan range

...  
...  
...  
...  
...  
...  

rdfs:subClassOf, Faculty, AssistProf  
rdfs:subClassOf, Faculty, AssocProf  
rdfs:subClassOf, Faculty, Professor  
...

Step 2: For each ?type in parallel,
POS – scan range

...  
rdf:type, AssistProf, Bob  
rdf:type, AssocProf, Amelia  
rdf:type, AssocProf, George  
rdf:type, Professor, Alice  
...

Enhancements ****
Batch Scanner

SELECT ?x WHERE {
    ?x takesCourse DBCourse .
    ?x rdf:type GraduateStudent .
}

Step 1: POS – scan range

| rdf:type, Professor, Alice |
| takesCourse, AICourse, John |
| takesCourse, AICourse, Zack |
| takesCourse, DBCourse, Bob |
| takesCourse, DBCourse, Greta |
| takesCourse, DBCourse, John |
| takesCourse, HCICourse, Alice |
| ... |

Step 2: batched for each ?x, SPO – index lookup

| ... |
| Bob, rdf:type, UndergradStudent |
| ... |
| Greta, rdf:type, GraduateStudent |
| ... |
| John, rdf:type, GraduateStudent |
| ... |

Enhancements ** ** **
Time Ranges

- SELECT ?load WHERE{
  ?measurement cpuLoad ?load.
  ?measurement timestamp ?ts.
  FILTER (?ts > "30 min ago")
}

- SELECT ?load WHERE{
  ?measurement cpuLoad ?load.
  ?measurement timestamp ?ts.
  timeRange (?ts, 1300, 1330)
}

Enhancements****
Outline

- Problem
- Background
- Rya
  - Triple index
  - Performance enhancements
- Experimental results
- Conclusions and future work
Experiments Set-up

- Accumulo 1.3.0
  - 1 Accumulo master
  - 10 Accumulo tablet servers
- Each node: 8 core Intel Xeon CPU, 16 GB RAM, 3 TB Hard Drive
- Tomcat server for Rya
- Java implementation
- Dataset: LUBM
Performance Metrics

- LUBM data set – 10 to 15000 universities
- Load time
- Queries per second
  - Using batch scanner
  - Without batch scanner
## Data Set - LUBM

<table>
<thead>
<tr>
<th>Nb Universities</th>
<th>Nb Triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1.3M</td>
</tr>
<tr>
<td>100</td>
<td>13.8M</td>
</tr>
<tr>
<td>1000</td>
<td>138.2M</td>
</tr>
<tr>
<td>2000</td>
<td>258.8M</td>
</tr>
<tr>
<td>5000</td>
<td>603.7M</td>
</tr>
<tr>
<td>10000</td>
<td>1.38B</td>
</tr>
<tr>
<td>15000</td>
<td>2.1B</td>
</tr>
</tbody>
</table>

Experiments *******
Load time

Experiments **********
Rya Query Performance - QpS

<table>
<thead>
<tr>
<th>#Univ</th>
<th>10</th>
<th>100</th>
<th>1K</th>
<th>2K</th>
<th>5K</th>
<th>10K</th>
<th>15K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>121.8</td>
<td>191.61</td>
<td>114.98</td>
<td>194.86</td>
<td>162.17</td>
<td>135.02</td>
<td>135.85</td>
</tr>
<tr>
<td>Q2</td>
<td>0.37</td>
<td>0.02</td>
<td>0.003</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.005</td>
</tr>
<tr>
<td>Q3</td>
<td>115.38</td>
<td>146.34</td>
<td>110.66</td>
<td>78.15</td>
<td>126.51</td>
<td>112.22</td>
<td>128.18</td>
</tr>
<tr>
<td>Q4</td>
<td>38.95</td>
<td>41.93</td>
<td>43.5</td>
<td>54.98</td>
<td>52.04</td>
<td>44.17</td>
<td>20.06</td>
</tr>
<tr>
<td>Q5</td>
<td>48.58</td>
<td>24.72</td>
<td>25.8</td>
<td>42.42</td>
<td>40.61</td>
<td>38.0</td>
<td>30.35</td>
</tr>
<tr>
<td>Q6</td>
<td>2.81</td>
<td>0.76</td>
<td>0.38</td>
<td>2.52</td>
<td>1.01</td>
<td>0.61</td>
<td>0.9</td>
</tr>
<tr>
<td>Q7</td>
<td>51.22</td>
<td>57.46</td>
<td>45.1</td>
<td>72.05</td>
<td>60.12</td>
<td>64.9</td>
<td>43.14</td>
</tr>
<tr>
<td>Q8</td>
<td>7.44</td>
<td>4.05</td>
<td>3.17</td>
<td>1.18</td>
<td>1.17</td>
<td>1.19</td>
<td>0.96</td>
</tr>
<tr>
<td>Q9</td>
<td>0.25</td>
<td>0.16</td>
<td>0.07</td>
<td>0.18</td>
<td>0.01</td>
<td>0.06</td>
<td>0.013</td>
</tr>
<tr>
<td>Q14</td>
<td>2.2</td>
<td>2.25</td>
<td>0.55</td>
<td>2.58</td>
<td>2.31</td>
<td>1.1</td>
<td>1.39</td>
</tr>
</tbody>
</table>

Experiments ********
Query 5

Experiments

- **Query 5**: Graph showing the performance of Q5 with and without a batch scanner.
Comparison with Other Systems

- Systems:
  - Graph Partitioning [HAR11]
  - SHARD [RS10]
- Benchmark: LUBM 2000

<table>
<thead>
<tr>
<th>System</th>
<th>Load Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHARD</td>
<td>10h</td>
</tr>
<tr>
<td>Graph Partitioning</td>
<td>4h 10min</td>
</tr>
<tr>
<td>Rya</td>
<td>3h 1min</td>
</tr>
</tbody>
</table>

Experiments **********
Comparison with Other Systems

Experiments

* * * * * * *

![Comparison with Other Systems Diagram]
Related Work

- RDF-3X [NW08] - centralized
- Graph Partitioning [HAR11] – graph partitioning + local RDF engines + MapReduce
- SHARD [RS10] – RDF triple store + HDFS
- Hexastore [WKB08] – six indexes
- SPARQL/MapReduce [MYL10] – MapReduce jobs to process SPARQL
Outline

- Problem
- Background
- Rya
  - Triple index
  - Performance enhancements
- Experimental results
- Conclusions and future work
Conclusions and Future Work

- **Rya** – scalable RDF Triple Store
  - Built on top of Accumulo and OpenRDF
  - Handles billions of triples
  - Millisecond query time for most queries

- **Future:**
  - Broader inferencing rules
  - New join algorithms
Thank You!

Questions?