Apache Rya: A Scalable RDF Triple Store

Adina Crainiceanu, Roshan Punnoose, David Rapp, Caleb Meier, Aaron Mihalik, Puja Valiyil, David Lotts, Jennifer Brown
RDF Data

- Very 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

Problem ** *****
Why RDF?

- W3C standard
- Large community/tool support
- Easy to understand
- Intrinsically represents a labeled, directed graph
  
  ![Diagram](image)

- Unstructured
  - Though with RDFS/OWL, can add structure

Problem ** ****
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
Apache Rya
-Distributed RDF Triple Store-

- Smartly store RDF data in Apache Accumulo
  - Scalability
  - Load balance
- Build on the RDF4J interface implementation for SPARQL
  - Fast queries
Outline

- Problem
- Background
- Rya
  - Triple index
  - Performance enhancements
  - Extra features
- Experimental results
- Conclusions and future work
RDF4J (OpenRDF Sesame)

- Utilities to parse, store, and query RDF data
- Supports SPARQL
- Ex: SELECT ?x WHERE {
  ?x worksAt USNA .
  ?x livesIn Baltimore .
}
- SPARQL queries evaluated based on triple patterns
  - Ex: (*, worksAt, USNA)
Apache Accumulo

- Google BigTable implementation

<table>
<thead>
<tr>
<th>Row ID</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Key</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Family</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Qualifier</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Visibility</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Timestamp</td>
<td></td>
</tr>
</tbody>
</table>

- 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
  - Additional features
- Experimental results
- Conclusions and future work
Architectural Overview - Rya

Query Processing

- Query Parsing
- Initial Query Execution Plan
- Query Execution

RDF4J

SAIL

Rya

SAIL

Data Storage

Accumulo
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
- Store graph name in the Column Family

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row ID</td>
<td></td>
</tr>
<tr>
<td>subject,predicate,object,type</td>
<td></td>
</tr>
<tr>
<td>graph name</td>
<td></td>
</tr>
<tr>
<td>visibility</td>
<td></td>
</tr>
<tr>
<td>timestamp</td>
<td></td>
</tr>
</tbody>
</table>
### Triple Table Index - Advantages

- Take advantage of native lexicographical sorting of row keys → fast range queries
- All patterns can be translated into a scan of one of these tables

<table>
<thead>
<tr>
<th>Row ID</th>
<th>Key</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>subject,predicate,object,type</td>
<td>graph name</td>
<td>visibility</td>
<td>timestamp</td>
</tr>
</tbody>
</table>

- A table with key and value columns for efficient range queries.
Sample Triple Storage

Example RDF triple:

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greta</td>
<td>worksAt</td>
<td>USNA</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>Greta, worksAt, USNA</td>
</tr>
<tr>
<td>POS</td>
<td>worksAt, USNA, Greta</td>
</tr>
<tr>
<td>OSP</td>
<td>USNA, Greta, worksAt</td>
</tr>
</tbody>
</table>
## Triple Patterns to Table Scans

<table>
<thead>
<tr>
<th>Triple Pattern</th>
<th>Table to Scan</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Greta, worksAt, USNA)</td>
<td>Any table (SPO default)</td>
</tr>
<tr>
<td>(Greta, worksAt, *)</td>
<td>SPO</td>
</tr>
<tr>
<td>(Greta, *, USNA)</td>
<td>OSP</td>
</tr>
<tr>
<td>(*, worksAt, USNA)</td>
<td>POS</td>
</tr>
<tr>
<td>(Greta, *, *)</td>
<td>SPO</td>
</tr>
<tr>
<td>(*, worksAt, *)</td>
<td>POS</td>
</tr>
<tr>
<td>(*, *, USNA)</td>
<td>OSP</td>
</tr>
<tr>
<td>(*, *, *)</td>
<td>any full table scan (SPO default)</td>
</tr>
</tbody>
</table>
Query Processing

SELECT ?x WHERE {
    ?x worksAt USNA .
    ?x livesIn Baltimore. }

Step 1: POS – scan range

... rdf:type, Woman, Elsa
worksAt, Cisco, John
worksAt, Cisco, Zack
worksAt, USNA, Bob
worksAt, USNA, Greta
worksAt, USNA, John
worksAt, UW, Elsa
...

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

... Bob, livesIn, Annapolis
... Greta, livesIn, Baltimore
... John, livesIn, Baltimore
...
More Complex Query Processing

SELECT ?x WHERE {
  ?x worksAt USNA.
  ?x livesIn Baltimore .
  ?x commuteMethod bike}

Step 1: POS – scan range

... rdf:type, Woman, Elsa
worksAt, Cisco, John
worksAt, Cisco, Zack
worksAt, USNA, Bob
worksAt, USNA, Greta
worksAt, USNA, John
worksAt, UW, Elsa
...

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

... ?x worksAt USNA

Bob, livesIn, Annapolis

... ?x livesIn Baltimore

Greta, livesIn, Baltimore

... ?x worksAt USNA

John, livesIn, Baltimore

Step 3: For each remaining ?x, SPO Table lookup

... ?x commuteMethod bike

Greta

... ?x commuteMethod, car

John

Rya * * * * * * * * *
Query Processing using Inference

SELECT ?x WHERE { ?x rdf:type Person }

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

Elsa
\[\text{rdf:type}\] Woman
\[\text{rdfs:subClassOf}\] Person

Rya ************
SELECT ?x WHERE {
    ?type rdfs:subClassOf Person.
    ?x rdf:type ?type .
}

**Step 1: POS – scan range**

| ... |
| ... |
| ... |
| ... |
| rdfs:subClassOf, Person, Child |
| rdfs:subClassOf, Person, Man |
| rdfs:subClassOf, Person, Woman |
| ... |
| ... |

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

| ... |
| ... |
| rdf:type, Child, Bob |
| rdf:type, Child, Jane |
| rdf:type, Child, Bob |
| rdf:type, Man, Adam |
| rdf:type, Man, George |
| rdf:type, Woman, Elsa |
| ... |
Inference Implementation

- **Step 1. Materialize inferred OWL model**
  - As RDF triples in Rya (refreshed when OWL model loaded/ changes)
    - Uses MapReduce jobs to infer the relationships or
  - As Blueprint graph in memory (refreshed periodically)
    - Uses TinkerPop Blueprints implementation

- **Step 2. Expand SPARQL query at runtime**
Challenges in Query Execution

- **Scalability and Responsiveness**
  - Massive amounts of data
  - Potentially large amounts of comparisons

Consider the Previous Example:

- Default query execution: comparing each “?x” returned from first statement pattern query to all subsequent triple patterns

**Poor query execution plans can result in simple queries taking minutes as opposed to milliseconds**
Outline

- Problem
- Background
- Rya
  - Triple index
  - Performance enhancements
  - Additional features
- Experimental results
- Conclusions and future work
Rya Query Optimizations

- **Goal:** Optimize query execution (joins) to better support real time responsiveness

- **Approaches:**
  - *Limit data in joins:* Use statistics to improve query planning
  - *Reduce the number of joins:* Materialized views
  - *Parallelize joins*
  - Accumulo Scanner /Batch Scanner use
  - Time Ranges

Enhancements *
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>livesIn</td>
<td>Predicate</td>
<td>5mil</td>
</tr>
<tr>
<td>Baltimore</td>
<td>Object</td>
<td>2.1mil</td>
</tr>
<tr>
<td>worksAt</td>
<td>Predicate</td>
<td>800K</td>
</tr>
<tr>
<td>USNA</td>
<td>Object</td>
<td>40K</td>
</tr>
</tbody>
</table>

```
SELECT ?x WHERE {
?x worksAt USNA.
?x livesIn Baltimore. }
```

Vs.

```
SELECT ?x WHERE {
?x livesIn Baltimore .
?x worksAt USNA }
```

Statistics * * * * * *
Rya Cardinality Usage

- Maintain cardinalities on the following triple patterns element combinations:
  - Single elements: Subject, Predicate, Object
  - Composite elements: Subject-Predicate, Subject-Object, Predicate-Object
- Computed periodically using MapReduce
  - Only store cardinalities above a threshold
- Only need to recompute cardinalities if the distribution of the data changes significantly

Statistics * * * * *
Limitations of Cardinality Approach

- Consider a more complicated query

```sql
SELECT ?x WHERE {
    ?x worksAt USNA.
    ?x commuteMethod bike.
    ?vehicle vehicleType SUV.
    ?x livesIn Baltimore.
    ?x owns ?vehicle.
}
```

- Cardinality approach does not take into account number of results returned by joins
- Solution lies in estimating the join selectivity for each pair of triples

Statistics

800K matches
20K matches
600K matches
800K matches
1 mil matches
254 mil matches
Using Join Selectivity

Query optimized using only Cardinality Info:

```
SELECT ?x WHERE {
  ?x worksAt USNA.
  ?x commuteMethod bike.
  ?vehicle vehicleType SUV.
  ?x livesIn Baltimore.
  ?x owns ?vehicle.
}
```

Query optimized using Cardinality and Join Selectivity Info:

```
SELECT ?x WHERE {
  ?x worksAt USNA.
  ?x commuteMethod bike.
  ?x livesIn Baltimore.
  ?x owns ?vehicle.
}
```

- Join selectivity measures number of results returned by joining two triple patterns.
- Due to computational complexity, estimate of join selectivity for triple patterns is pre-computed and stored in Accumulo.

Statistics *** *** ***
Join Selectivity: General

- For statement patterns $<$?x, p_1, o_1$>$ and $<$?x, p_2, o_2$>$,

\[ Sel(<x, p_1, o_1 > \land <x, p_2, o_2 >) \]

\[ \approx \min\{Sel(<x, p_1, o_1 > \land <x, y, z >), Sel(<x, p_2, o_2 > \land <x, y, z >)\} \]

- Full table join statistics precomputed and stored in index
- Join statistics for each triple pattern computed using:

\[ Sel(<x, p_1, o_1 > \land <x, y, z >) = \sum_{<c, p_1, o_1>} \left| \frac{<c, y, z >}{<x, p_1, o_1 > \land <x, y, z >} \right| \]

- Use analogous definition if variables appear in predicate or object position
- Approach based on RDF-3X [NW08]
Use Join Selectivity in Rya

- Greedy approach: start with most selective triple pattern and add patterns based on minimization of a cost function
  \[ C = \text{leftCard} + \text{rightCard} + \text{leftCard} \times \text{rightCard} \times \text{selectivity} \]
  - C measures number of entries Accumulo must scan and the number of comparisons required to perform the join
- Selectivity set to one if two triple patterns share no common variables, otherwise precomputed estimates used
  - Ensures that patterns with common variables are grouped together

Statistics ★★★★★★★
Pre-Computed Joins

- Reduce number of joins by pre-computing common joins

```sparql
SELECT ?x WHERE {
  ?x worksAt USNA.
  ?x commuteMethod bike.
  ?x livesIn Baltimore.
  ?x owns ?vehicle.
  ?vehicle vehicleTypeType SUV.
}
```

Pre-compute using batch processing and look up during query execution.
Using Pre-Computed Joins

1. Pre-compute a portion of the query using MapReduce
2. Store SPARQL describing the query along with pre-computed values in Accumulo
3. Normalize query variables to match stored SPARQL variables during query execution

Stored SPARQL

```
SELECT ?x WHERE {
  ?x worksAt USNA.
  ?x commuteMethod bike.
  ?x livesIn Baltimore.
  ?x owns ?vehicle.
  ?vehicle vehicleType SUV.
}
```

Index Result Table

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Aaron</td>
<td>ToyotaRav4</td>
</tr>
<tr>
<td>Caleb</td>
<td>JeepCherokee</td>
</tr>
<tr>
<td>Puja</td>
<td>HondaCRV</td>
</tr>
<tr>
<td></td>
<td>....</td>
</tr>
</tbody>
</table>
Parallel Joins

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

Step 1: POS – scan range

```
...  
...  
...  
...  
...  
rdfs:subClassOf, Person, Child  
rdfs:subClassOf, Person, Man  
rdfs:subClassOf, Person, Woman
...  
...  
```

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

```
...  
...  
rdf:type, Child, Bob  
rdf:type, Child, Jane
...  
rdf:type, Man, Adam  
rdf:type, Man, George
...  
rdf:type, Woman, Elsa
...  
...  
```

|| Joins *
Batch Scanner

SELECT ?x WHERE {
  ?x worksAt USNA .
  ?x livesIn Baltimore .
}

Step 1: POS – scan range

- rdf:type, Woman, Elsa
- worksAt, Cisco, John
- worksAt, Cisco, Zack
- worksAt, USNA, Bob
- worksAt, USNA, Greta
- worksAt, USNA, John
- worksAt, UW, Elsa

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

- Bob, livesIn, Annapolis
- Greta, livesIn, Baltimore
- John, livesIn, Baltimore

Result: Decreases network connections by up to 1K fold
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)
}

Result: Allow RDF querying on a small subset of data (based on loading time)
Additional Features

- Range queries support in serialized format for many types
- Regular expression filter incorporated into Accumulo scan
- Support for named graphs
- SPARQL to Pig translation
- MongoDB back-end support
- Entity-centric index
- Temporal, geospatial, full-text indexing

Additions *
Outline

- Problem
- Background
- Rya
  - Triple index
  - Performance enhancements
  - Additional features
- 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

Experiments

Experiments
## 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

Number of Universities

Load Time (minutes)
## 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>
Query 5

Experiments

Graph showing the number of queries per second over the number of universities with and without a batch scanner.
Query Optimization Results

Ran 14 queries against the Lehigh University Benchmark (LUBM) dataset (33.34 million triples)
- LUBM queries 2, 5, 9, and 13 were discarded after 3 runs due to query complexity
- Remaining queries were executed 12 times
- Cluster Specs:
  - 8 worker nodes, each has 2 x 6-Core Xeon E5-2440 (2.4GHz) Processors and 48 GB RAM
- Results indicate that cardinality and join selectivity optimizations provide improved or comparable performance

<table>
<thead>
<tr>
<th>Query</th>
<th>Rya (s)</th>
<th>Rya Cardinality (s)</th>
<th>Rya Join Selectivity (s)</th>
<th># of Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUBM 1</td>
<td>37.93</td>
<td>0.05</td>
<td>0.19</td>
<td>4</td>
</tr>
<tr>
<td>LUBM 3</td>
<td>102.86</td>
<td>0.10</td>
<td>0.33</td>
<td>6</td>
</tr>
<tr>
<td>LUBM 4</td>
<td>38.41</td>
<td>38.18</td>
<td>40.49</td>
<td>35</td>
</tr>
<tr>
<td>LUBM 6</td>
<td>13.06</td>
<td>13.74</td>
<td>13.26</td>
<td>1978437</td>
</tr>
<tr>
<td>LUBM 7</td>
<td>NA</td>
<td>291.95</td>
<td>0.91</td>
<td>59</td>
</tr>
<tr>
<td>LUBM 8</td>
<td>NA</td>
<td>NA</td>
<td>4.70</td>
<td>5916</td>
</tr>
<tr>
<td>LUBM 10</td>
<td>103.75</td>
<td>0.09</td>
<td>0.17</td>
<td>0</td>
</tr>
<tr>
<td>LUBM 11</td>
<td>53.10</td>
<td>0.55</td>
<td>0.65</td>
<td>224</td>
</tr>
<tr>
<td>LUBM 12</td>
<td>0.44</td>
<td>0.49</td>
<td>0.82</td>
<td>0</td>
</tr>
<tr>
<td>LUBM 14</td>
<td>13.13</td>
<td>13.30</td>
<td>13.11</td>
<td>1978437</td>
</tr>
</tbody>
</table>
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
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
  - Additional features
- 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
  - Apache project (incubating)

- Future:
  - New join algorithms
  - Federated Rya
  - Improved MongoDB support
  - Spark support
  - Temporal and spatial indexing
Rya Community – Join Us!

- Friendly
- Responsive
- Growing

How you can help:
- Join the dev list, participate in discussions
- Try the software
- Submit bug reports, new features requests
- Improve documentation
- Verify release candidates
Get Involved!

Apache Rya (incubating)

https://rya.apache.org

dev@rya.incubator.apache.org
Thank You!

Questions?