Apache Rya: A Scalable RDF Triple Store

Adina Crainiceanu, Roshan Punnoose, David Rapp,

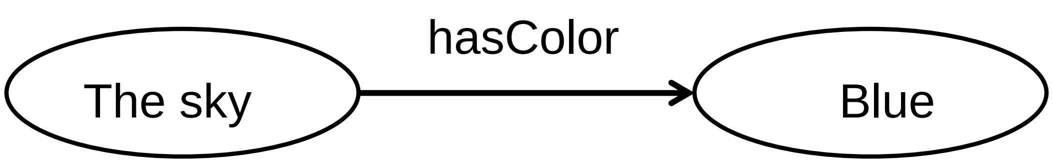
Caleb Meier, Aaron Mihalik, Puja Valiyil, David Lotts, Jennifer Brown

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

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

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

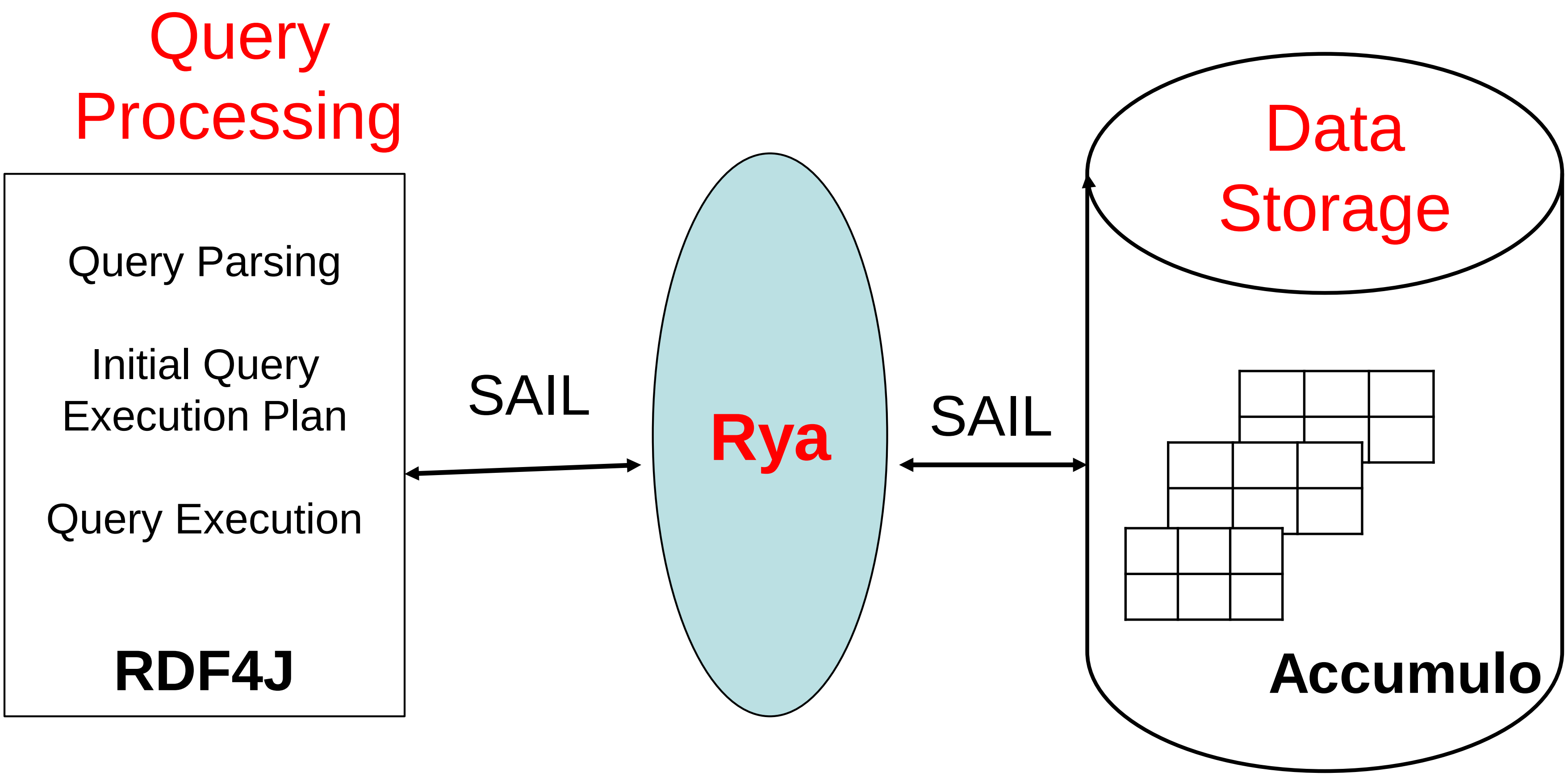
Key					Value
Row ID	Column			Timestamp	
	Family	Qualifier	Visibility		

  - 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



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

Key					*Value*
Row ID	*Column*			*Timestamp*	
	Family	*Qualifier*	*Visibility*		
subject,predicate,object,type	graph name		visibility	timestamp	

Triple Table Index - Advantages

Key					Value
Row ID	*Column*			*Timestamp*	
	Family	*Qualifier*	*Visibility*		
subject,predicate,object,type	graph name		visibility	timestamp	

- 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

Sample Triple Storage

Example RDF triple:

Subject	Predicate	Object
Greta	worksAt	USNA

Stored RDF triple in Accumulo tables:

Table	Stored Triple
SPO	Greta, worksAt, USNA
POS	worksAt, USNA, Greta
OSP	USNA, Greta, worksAt

Triple Patterns to Table Scans

Triple Pattern	Table to Scan
(Greta, worksAt, USNA)	Any table (SPO default)
(Greta, worksAt, *)	SPO
(Greta, *, USNA)	OSP
(*, worksAt, USNA)	POS
(Greta, *, *)	SPO
(*, worksAt, *)	POS
(*, *, USNA)	OSP
(*, *, *)	any full table scan (SPO default)

Query Processing

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

Step 1: POS – scan range

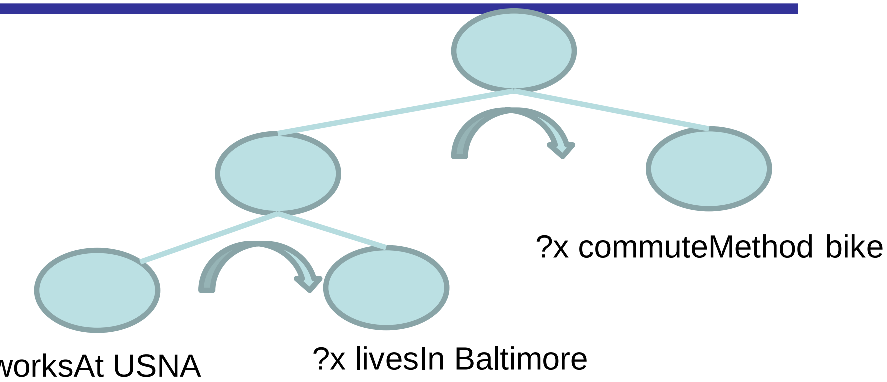
…
rdf:type, Woman, Elsa
worksAt, Cisco, John
worksAt, Cisco, Zack
worksAt, USNA, Bob
worksAt, USNA, Greta
worksAt, USNA, John
worksAt, UW, Elsa
…

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

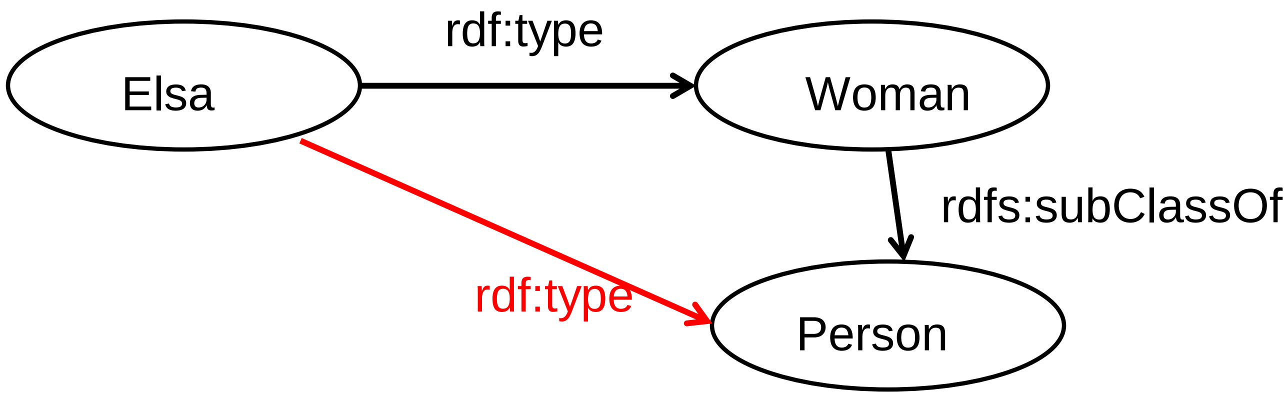
…
~~Bob, livesIn, Annapolis~~
…
Greta, livesIn,Baltimore
…
John, livesIn, Baltimore
…

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

…
Greta, commuteMethod, bike
…
~~John, commuteMethod, car~~
…

Query Processing using Inference

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



New query: SELECT ?x WHERE {

?type rdfs:subClassOf Person .

?x rdf:type ?type }

Query Plan for Expanded Query

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, 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:

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

- 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

Optimized Joins with Statistics

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

Value	Role	Cardinality
livesIn	Predicate	5mil
Baltimore	Object	2.1mil
worksAt	Predicate	800K
USNA	Object	40K

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

Vs.

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

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

Limitations of Cardinality Approach

- Consider a more complicated query

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

?x worksAt USNA.	20K matches
?x commuteMethod bike.	600K matches
?vehicle vehicleType SUV.	800K matches
?x livesIn Baltimore.	1 mil matches
?x owns ?vehicle.	254 mil matches

- 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

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.
  ?vehicle vehicleType SUV.  }
```

- 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

Join Selectivity: General

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

$$Sel(<?x, p_1, o_1 > \bowtie <?x, p_2, o_2 >)$$
$$\approx \min\{Sel(<?x, p_1, o_1 > \bowtie <?x, ?y, ?z >), Sel(<?x, p_2, o_2 > \bowtie <?x, ?y, ?z >)\}$$

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

$$Sel(<?x, p_1, o_1 > \bowtie <?x, ?y, ?z >) = \frac{\sum_{<c, p_1, o_1>} | < c, ?y, ?z > |}{| <?x, p_1, o_1 > || <?x, ?y, ?z > |}$$

- 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 = leftCard + rightCard + leftCard*rightCard*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

Pre-Computed Joins

- Reduce number of joins by pre-computing common joins
  - Approach based on: Heese, Ralf, et al. *"Index Support for SPARQL." European Semantic Web Conference, Innsbruck, Austria. 2007.*

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

Pre-compute using batch processing and look up during query execution

Views * *

Using Pre-Computed Joins

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

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 ?person ?car
WHERE {
?person livesIn Baltimore.
?person owns ?car.
?car vehicleType SUV.
}

Index Result Table
....
Aaron, ToyotaRav4
Caleb, JeepCherokee
Puja, HondaCRV
....

Views * *

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
…

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

Scanner *

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

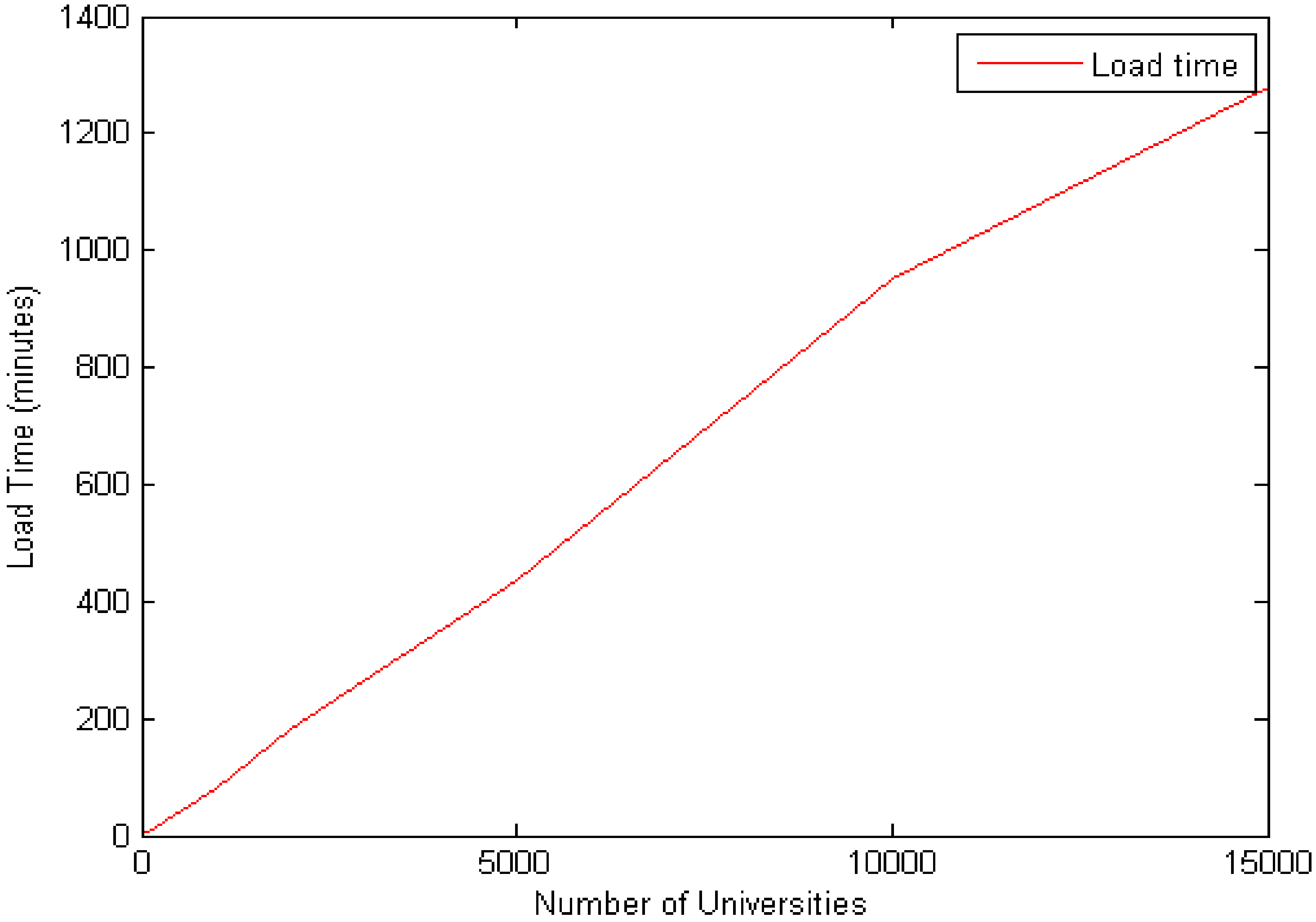
Performance Metrics

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

Data Set - LUBM

Nb Universities	Nb Triples
10	1.3M
100	13.8M
1000	138.2M
2000	258.8M
5000	603.7M
10000	1.38B
15000	2.1B

Load time

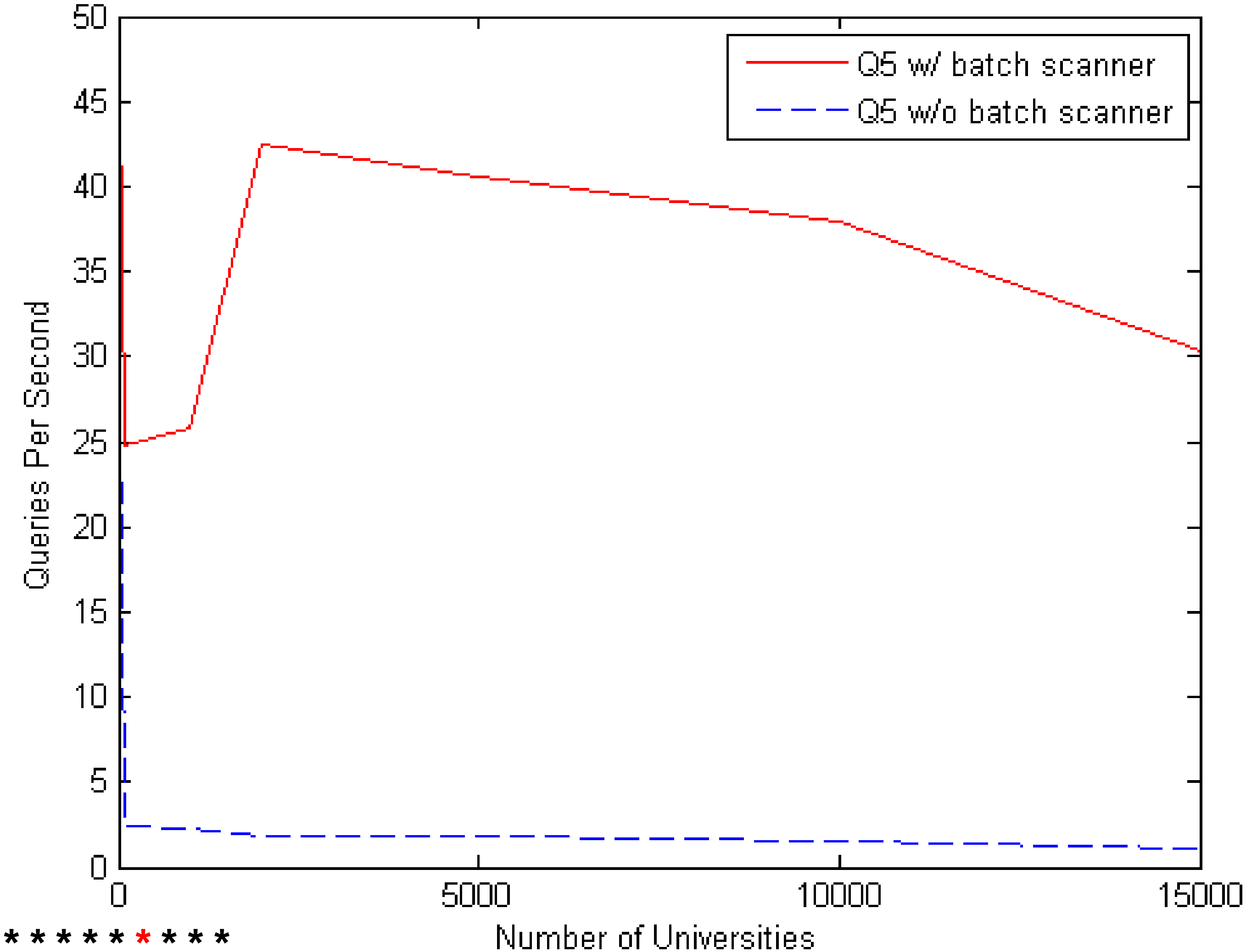


Experiments * * * * * * * * *

Rya Query Performance - QpS

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

Query 5


Q5 w/ batch scanner
Q5 w/o batch scanner
Queries Per Second
Number of Universities
0
5
10
15
20
25
30
35
40
45
50
0
5000
10000
15000

Query Optimization Results

Query	Rya (s)	Rya Cardinality (s)	Rya Join Selectivity (s)	# of Results
LUBM 1	37.93	0.05	0.19	4
LUBM 3	102.86	0.10	0.33	6
LUBM 4	38.41	38.18	40.49	35
LUBM 6	13.06	13.74	13.26	1978437
LUBM 7	NA	291.95	0.91	59
LUBM 8	NA	NA	4.70	5916
LUBM 10	103.75	0.09	0.17	0
LUBM 11	53.10	0.55	0.65	224
LUBM 12	0.44	0.49	0.82	0
LUBM 14	13.13	13.30	13.11	1978437

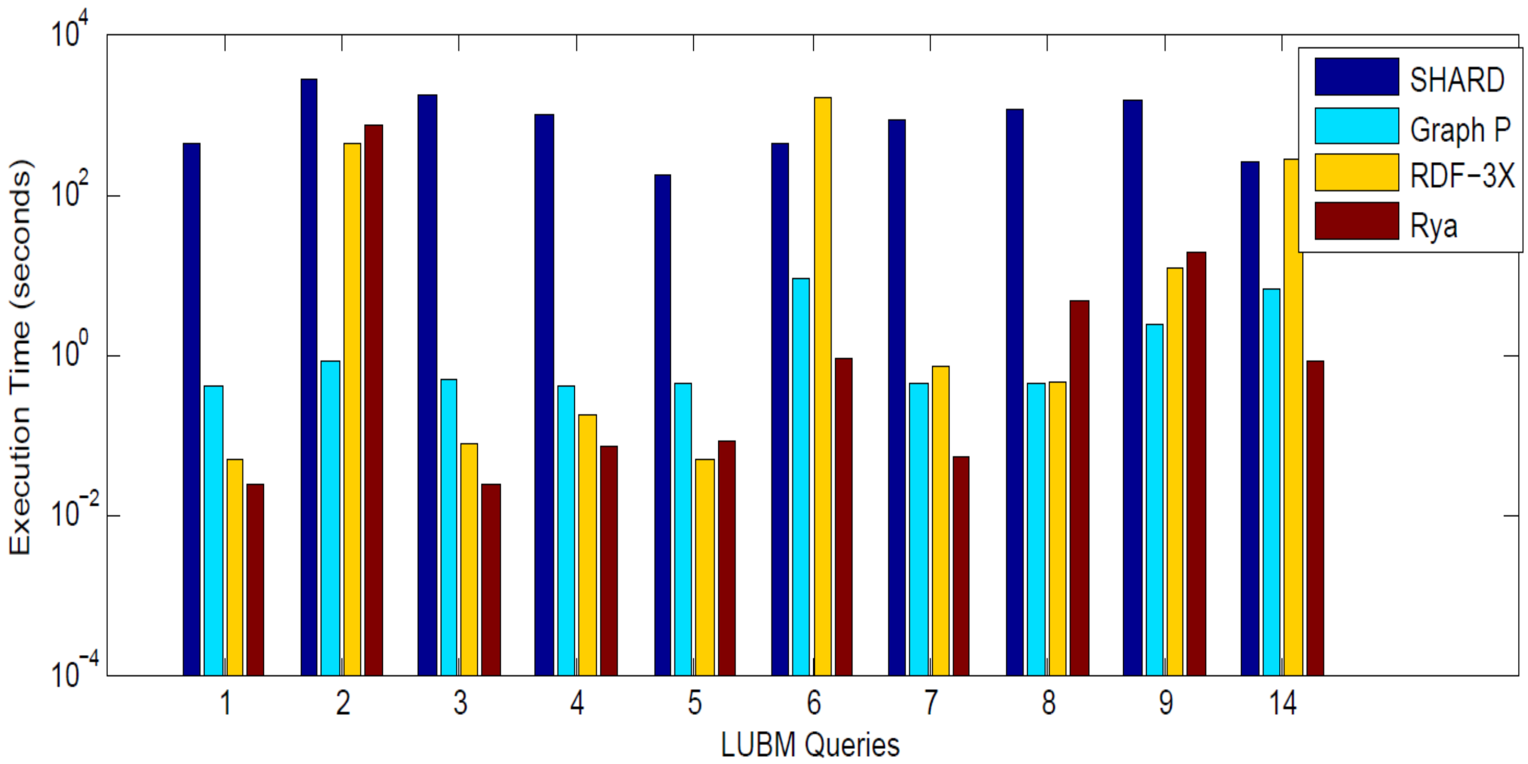
- Ran 14 queries against the Lehigh University Benchmark (LUBM) dataset (33.34 million triples)
  - LUBM queries 2, 5, 9, and 13 were discarded after 3 runs due to query complexity
    - Remaining queries were executed 12 times
  - Cluster Specs:
    - 8 worker nodes, each has 2 x 6-Core Xeon E5-2440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

Comparison with Other Systems

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

System	Load Time
SHARD	10h
Graph Partitioning	4h 10min
Rya	3h 1min

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?