

# Rya: A Scalable RDF Triple Store for the Clouds

Roshan Punnoose  
Proteus Technologies  
roshanp@gmail.com

Adina Crainiceanu  
US Naval Academy  
adina@usna.edu

David Rapp  
Laboratory for Telecommunication  
Sciences  
rapp@ltsnet.net

## ABSTRACT

Resource Description Framework (RDF) was designed with the initial goal of developing metadata for the Internet. While the Internet is a conglomeration of many interconnected networks and computers, most of today's best RDF storage solutions are confined to a single node. Working on a single node has significant scalability issues, especially considering the magnitude of modern day data. In this paper we introduce a scalable RDF data management system that uses Accumulo, a Google Bigtable variant. We introduce storage methods, indexing schemes, and query processing techniques that scale to billions of triples across multiple nodes, while providing fast and easy access to the data through conventional query mechanisms such as SPARQL. Our performance evaluation shows that in most cases, our system outperforms existing distributed RDF solutions, even systems much more complex than ours.

**Categories and Subject Descriptors:** H.3.2 Information Storage, H.3.3 Information Search and Retrieval, H.3.4 Systems and Software - Distributed Systems H.2.4 Systems - Distributed Databases, Query Processing

**General Terms:** Algorithms, Management, Performance.

**Keywords:** RDF triple store, distributed, scalable.

## 1. INTRODUCTION

The Resource Description Framework (RDF) [14] is a family of W3C specifications traditionally used as a metadata data model, a way to describe and model information, typically of the World Wide Web. In the most fundamental form, RDF is based on the idea of making statements about resources in the form of <subject, predicate, object> expressions called *triples*. To specify the title of the main US Naval Academy web page, one could write the triple <USNA Home, :titleOf, http://www.usna.edu/homepage.php>. As RDF is meant to be a standard for describing the Web resources, a large and ever expanding set of data, methods must be devised to store and retrieve such a large data set.

While very efficient, most existing RDF stores [19, 5, 11, 8, 2] rely on a centralized approach, with one server running very specialized hardware. With the tremendous increase in data size, such solutions will likely not be able to scale up.

With improvements in parallel computing, new methods can be devised to allow storage and retrieval of RDF across large compute

clusters; this allows handling data of unprecedented magnitude.

In this paper, we propose *Rya*, a new scalable system for storing and retrieving RDF data in a cluster of nodes. We introduce a new serialization format for storing the RDF data, an indexing method to provide fast access to data, and query processing techniques for speeding up the evaluation of SPARQL queries. Our methods take advantage of the storing, sorting, and grouping of data that Accumulo [1, 13] provides. We show through experiments that our system scales RDF storage to billions of records and provides millisecond query times.

## 2. BACKGROUND

### 2.1 Accumulo

Accumulo [1, 13] is an open-source, distributed, column-oriented store modeled after Google's Bigtable [3]. Accumulo provides random, realtime, read/write access to large datasets atop clusters of commodity hardware. Accumulo leverages Apache Hadoop Distributed File System [4], the open source implementation of the Google File System [16]. In addition to Google Bigtable features, Accumulo features automatic load balancing and partitioning, data compression, and fine grained security labels [13].

Accumulo is essentially a distributed key-value store that provides sorting of keys in lexicographical ascending order. Each key is composed of (Row ID, Column, Timestamp) as shown in Table 1. Rows in the table are stored in contiguous ranges (sorted by key) called tablets, so reads of short ranges are very fast. Tablets are managed by tablet servers, with a tablet server running on each node in a cluster. Section 3 describes how the locality properties provided by sorting of the Row ID is used to provide efficient lookups of triples in Rya.

Key				Timestamp	Value
Row ID	Column				
	Family	Qualifier	Visibility		

Table 1: Accumulo Key-Value

We chose Accumulo as the backend persistence layer for Rya over other notable Google BigTable variants such as HBase [6] because it provides a few extra important features. First, Accumulo provides a server side *Iterator* model that helps increase performance by performing large computing tasks directly on the servers and not on the client machine, thus avoiding the need to send large amounts of data across the network. Second, Accumulo provides a simple cell level security mechanism for clients that are interested in such fine grained security. Triples loaded with specific security labels can also be queried with the same labels. Third, Accumulo

also provides a *Batch Scanner* client API that allows us to reduce multiple range scans into one merged client request. The Batch Scanner is an efficient method for processing a large number of range scans quickly; it does this by merging all the scans into the smallest number of scans possible to return the correct data set. See Section 5.2 for more information. Fourth, Accumulo provides an efficient method of *Bulk Importing* large data sets to reduce ingest (data loading) time. The Bulk Import process uses Map Reduce to move rows to the correct servers and perform a direct import locally on the servers reducing ingest throughput significantly. Fifth, Accumulo provides native Bloom filters to increase the performance of row based lookups.

## 2.2 RDF, OWL, and SPARQL

RDF is a standard used in describing resources on the World Wide Web. The smallest data unit is a triple consisting of: subject, predicate, and object. Using this framework, it is very easy to describe any resource. In general, RDF is an open world framework that allows anyone to make any statement about any resource. For example, to say that S0 is a graduate student, the simplified version of the RDF triple is <S0, rdf:type, ub:GraduateStudent>.

The Web Ontology Language (OWL) is a framework for describing models or 'ontologies' for RDF. It defines concepts, relationships, and/or structure of RDF documents. These models can be used to 'reason' information about entities within a given domain. For example, the OWL model could define the predicate *hasMother* to be equivalent to *hasMom*. Then if a triple <Bob, hasMother, Rose> is defined, it is also 'reasoned' that <Bob, hasMom, Rose>.

SPARQL is an RDF query language. Similar with SQL, SPARQL has SELECT and WHERE clauses; however, it is based on querying and retrieving RDF triples. The WHERE clause typically contains a set of *triple patterns*. Unlike RDF triples, the triple patterns can contain variables. For example, the SPARQL query to find all the graduate students in some data set would be: *SELECT ?x WHERE { ?x rdf:type ub:GraduateStudent }*.

## 2.3 OpenRDF Sesame Framework

OpenRDF Sesame [12] is a well-known framework for processing RDF data. The framework provides out of the box utilities to parse, store, and query RDF data. Sesame currently fully supports the SPARQL 1.1 query language. In addition, it provides utilities to parse a variety of triple formats: RDF/XML, NTriples, N3, etc. The framework also provides an extension API, the SAIL API, to plug in a custom RDF persistence layer. Implementing the SAIL (Storage and Inference Layer) API can be as simple as providing a method to store a triple and a method to query based on a triple pattern. The framework provides the wrapper layers to parse various triple formats and convert high level query languages, such as SPARQL, into simple triple patterns.

However, if desired, the SAIL API provides interfaces to work more deeply with the query by interacting directly with the *Execution Plan*. The Execution Plan defines the order of execution, the results returned, where joins are to occur, etc. Here is a sample Execution Plan from the SAIL API given a SPARQL query:

SPARQL Query:

```
SELECT ?x
WHERE {
    ?x rdf:type ub:GraduateStudent .
    ?x ub:takesCourse <.../GraduateCourse2> .
} limit 10
```

Execution Plan:

```
QueryRoot
```

```
Slice ( limit=10 )
Projection
  ProjectionElemList
    ProjectionElem "x"
Join
  StatementPattern
    Var (name=x)
    Var (value=rdf:type)
    Var (value=ub:GraduateStudent)
  StatementPattern
    Var (name=x)
    Var (value=ub:takesCourse)
    Var (value=http://.../GraduateCourse2)
```

In addition to working directly with the triple patterns (StatementPattern in the above plan) provided by the Execution Plan, the Rya query engine also parses the Execution Plan directly to perform various operations. For example, it reorders joins to achieve better performance, or expands triple patterns to implement inferencing.

This paper introduces Rya, an RDF triple store that utilizes the OpenRDF Sesame SAIL API to create a custom, pluggable, RDF persistence layer to store and query RDF triples from Accumulo.

## 3. RYA: STORING AND RETRIEVING RDF

### 3.1 Three Table Index

An RDF triple contains a subject, predicate, and object. Our solution is based on the fact that Accumulo sorts and partitions all key-value pairs based on the Row ID part of the key. This means that as the data grows, rows will be grouped and sorted based on the lexicographical sort of the Row ID, providing very fast read/write access to short ranges across a large data set. Rows with similar IDs will be grouped into the same tablet/server for faster access. In addition, Accumulo will divide large groups based on a configuration parameter and move them to separate servers.

Designing the table layout in Accumulo requires a thorough understanding of how the data will be queried. Using the SAIL interface to query Rya, triples are queried using triple patterns. An example of triple pattern may be (subject, predicate, \*), which requires that the persistence layer returns all triples that match the given subject and predicate.

We propose a new method of storing triples in Accumulo, by indexing triples across three separate tables to satisfy all the permutations of the triple pattern. These tables store the triple in the Accumulo Row ID and order the subject, predicate, object differently for each table. This solution utilizes the row-sorting scheme of Accumulo to efficiently store and query triples across multiple Accumulo tablets, in effect creating three clustered indexes for the data. The three tables are SPO, POS, and OSP, and are named based on the components order of the triples stored. The SPO table stores a triple in the Row ID as (Subject, Predicate, Object), the POS table as (Predicate, Object, Subject), and the OSP table as (Object, Subject, Predicate). While there are six possible permutations of the triple components (subject, predicate, object), three of these permutations are sufficient and necessary to efficiently answer each possible triple pattern by using only a range scan. Table 2 shows how all eight types of the triple patterns can map to an Accumulo range scan of one of the three tables.

All the data for the triple resides in the Accumulo Row ID. This offers several benefits: 1) by using a direct string representation, we can do direct range scans on the literals; 2) the format is very easy to serialize and deserialize, which provides for faster query

Triple Pattern	Accumulo Table to Scan
(subject, predicate, object)	SPO
(subject, predicate, *)	SPO
(subject, *, object)	OSP
(*, predicate, object)	POS
(subject, *, *)	SPO
(*, predicate, *)	POS
(*, *, object)	OSP
(*, *, *)	SPO (full table scan)

**Table 2: Triple Patterns Mapped to Table Scans**

and ingest; 3) since no information needs to be stored in the Column Family, Qualifier, or Value fields of the Accumulo tables, the storage requirements for the triples are significantly reduced. Section 3.2 describes concrete examples of how these triples are stored and queried.

### 3.2 Sample Storage and Querying of Triples

We show through an example how a triple is stored in Rya and how queries are processed.

Table 3 shows an example triple taken from the LUBM [9] dataset. The triple expresses the fact that a particular professor (subject) earned his/her degree (predicate) from a given university (the object). Table 4 shows how the triple is stored in the three table indexes SPO, POS, and OSP. For easy of reading, we use comma as separator in the examples, but the Unicode null character `\u0000` is used in practice. In each table, the triple is stored in the Row ID part of the table, and the three components of the triple are concatenated in the order corresponding to the particular table.

Subject	Predicate	Object
<code>http://Univ0/Professor3</code>	<code>urn:degreeFrom</code>	<code>http://Univ2</code>

**Table 3: Sample RDF Triple**

Table	Stored Triple
SPO	<code>http://Univ0/Professor3,urn:degreeFrom,http://Univ2</code>
POS	<code>urn:degreeFrom,http://Univ2,http://Univ0/Professor3</code>
OSP	<code>http://Univ2,http://Univ0/Professor3,urn:degreeFrom</code>

**Table 4: Sample RDF Triple in Rya**

**Example Query:** Consider the query: *Find all Graduate Students that take the course identified by `ub:U0:C0`.* The corresponding SPARQL query is

```
SELECT ?x
WHERE {
  ?x ub:takesCourse ub:U0:C0 .
  ?x rdf:type ub:GraduateStudent
}
```

The query evaluation plan created by the SAIL API is shown below:

```
QueryRoot
  Projection
    ProjectionElemList
      ProjectionElem "x"
    Join
```

```
StatementPattern
  Var (name=x)
  Var (value=ub:takesCourse)
  Var (value=ub:U0:C0)
StatementPattern
  Var (name=x)
  Var (value=rdf:type)
  Var (value=ub:GraduateStudent)
```

The triple patterns generated are `(*,ub:takesCourse,ub:U0:C0)` and `(*,rdf:type,ub:GraduateStudent)`. The query plan executed by Rya's Query Evaluation Engine is:

Step 1: Accumulo Range Query

- Table: POS
- Scan Start Value: `ub:takesCourse,ub:U0:C0`
- Scan End Value: `ub:takesCourse,ub:U0:C0`
- Result: Return all subjects (`?x`)

Step 2: Accumulo Equality Query (For each "`?x`" from Step 1)

- Table: SPO
- Scan Value: `?x,rdf:type,ub:GraduateStudent`, where `?x` is bound to the value found in Step 1
- Result: Does row exist?

This SPARQL query is evaluated using an index nested loops join where Step 1 scans for all rows starting with "`ub:takesCourse,ub:U0:C0`" in the POS table. For each result found in Step 1, an index lookup is performed in Step 2 to find those results (`?x`) that are of `rdf:type ub:GraduateStudent`.

## 4. QUERY PROCESSING

One of RDF's strengths is the ability to 'infer' relationships or properties. Rya supports `rdfs:subClassOf`, `rdfs:subPropertyOf`, and `owl:EquivalentProperty` inferences. We describe below our methods for inferencing and query processing.

### 4.1 Query Planner Stage 1

The first stage of the query inferencing is performed only once, and deals with creating explicit relationships based on the implicit relationships defined by an OWL model, the standard of defining inferred relationships. For example, given the following explicit relationships: *FullProfessor* (*subClassOf*) *Professor* and *Professor* (*subClassOf*) *Faculty*, the implicit relationship *FullProfessor* (*subClassOf*) *Faculty* exists.

While some of the relationships are defined explicitly in the OWL model, the work of inferencing is to find the relationships that are not explicitly defined.

In Rya, we express an OWL model as a set of triples and store them in the triple store. One of the benefits of storing all the data in the triple store is that Hadoop MapReduce can be utilized to run large batch processing jobs against the data set.

The first stage of the process is performed only once, at the time the OWL model is loaded into Rya. Stage 1 consists of running a MapReduce job to iterate through the entire graph of relationships and output the implicit relationships found as explicit RDF triples stored into the RDF store. The amount of time taken to complete Stage 1 is directly related to the depth of the tree of relationships in the original OWL model. In our use cases, we tested with trees as large as 5 levels deep. Each level requires a separate iteration of

the MapReduce phase, each taking about 30 seconds to complete. Since this stage is only applied to the OWL model relationships, and not to the actual RDF data that matches the model, the size of the data passed between iterations depends only on the size and depth of the model. In practice, the number of triples to represent the data model is a very small fraction of the data stored.

## 4.2 Query Planner Stage 2

The second stage of the process is performed every time a query is run. Once all the explicit and implicit relationships are stored in Rya, the Rya query planner is able to expand the query at runtime to utilize all these relationships. The expansion technique is applied to "rdf:type", "subPropertyOf" and "EquivalentProperty" inferences. For example, consider the following SPARQL query:

```
SELECT ?prof
WHERE {
  ?prof rdf:type ub:Faculty }
```

Based on the data generated in Stage 1, the OWL model for the ontology used by the query specifies that FullProfessor is a "subClassOf" Professor, and Professor is "subClassOf" Faculty, and implies that FullProfessor is "subClassOf" Faculty. When the query asks for all triples with "rdf:type" Faculty, it is really asking for all triples that have "rdf:type" FullProfessor or Professor or Faculty, based on the "subClassOf" relationships.

The query evaluation plan generated by the SAIL API for the above query is given below:

```
QueryRoot
  Projection
    ProjectionElemList
      ProjectionElem "prof"
    StatementPattern
      Var (name=prof)
      Var (value=rdf:type)
      Var (value=ub:Faculty)
```

The query planner in Rya traverses the query evaluation plan produced by the SAIL API, searching for StatementPattern nodes that contain "rdf:type". The Rya query planner has built-in rules to expand the "rdf:type" predicate as a separate join, part of the same query, using "rdfs:subClassOf" and "rdf:type" predicates. The resulting expanded query evaluation plan is given below:

```
QueryRoot
  Projection
    ProjectionElemList
      ProjectionElem "prof"
    Join
      StatementPattern
        Var (name=type)
        Var (value=rdfs:subClassOf)
        Var (value=ub:Faculty)
        Var (name=subclass_expanded)
      StatementPattern
        Var (name=prof)
        Var (value=rdf:type)
        Var (name=type)
        Var (name=subclass_expanded)
```

The first triple pattern (?type, rdfs:subClassOf, ub:Faculty) asks for all "subClassOf" Faculty relationships, which returns ?type in

{FullProfessor, Professor, Faculty}. To evaluate the second triple pattern, (?prof, rdf:type, ?type), three parallel queries are run to find all subjects that have "rdf:type" equal to the value of ?type. This query returns all subjects that have "rdf:type" in {FullProfessor, Professor, Faculty}. In addition to the query expansion, we add a marker on each StatementPattern, *subclass-expanded*, to identify the nodes that have already been expanded, so query expansion is not applied again.

## 5. PERFORMANCE ENHANCEMENTS

The first iteration of Rya simply made use of the three table indexes to perform scans and return results. This approach was very fast and worked well in many real situations; however, the method can be improved using an array of key techniques that we describe below.

### 5.1 Parallel Joins

The OpenRDF Sesame framework provides a default "Evaluation Strategy" that executes the query plan created by the framework for an input SPARQL query. However, much of the query processing is done in a single thread. For example, consider the question of finding the students that take a particular course Course0 which translates as the following join between two triple patterns:

```
TQ1: (?x, takesCourse, Course0)
TQ2: (?x, type, Student)
```

The default join algorithm executes first the TQ1 query which may return ?x in {Alex, Bob, Rachel}. The next execution step creates an Accumulo scan for each of these answers to check if they are "type, Student". For example, a scan will first check the existence of the triple (Alex, type, Student), then another scan for (Bob, type, Student), then for (Rachel,type, Student). The default implementation is sequential, all in one thread. By executing the joins in parallel, we improve performance up to 10-20x the normal speed. We achieve parallelization by utilizing the Java concurrent library to submit tasks. The default size of the thread pool associated with the parallelization of joins is set to 10. In our testing, this setting depends entirely on the processor speed, number of cores, and the type of query.

### 5.2 Batch Scanner

Consider again the example of finding students that take Course0, which requires the join between two triple patterns:

```
TQ1: (?x, takesCourse, Course0)
TQ2: (?x, type, Student)
```

As described above, the join algorithm executes first TQ1, and then for each result returned, runs TQ2 with (?x) bound to a value returned by TQ1. For a query that is not very selective, TQ1 may return a large number of answers, say 5000; for each row returned, a new Accumulo Scanner is instantiated and invoked to check whether the returned answer satisfies TQ2. This process consumes network resources, and can be improved by introducing the Accumulo Batch Scanner. The Batch Scanner is designed to efficiently scan many ranges in Accumulo. For example, an Accumulo cluster may have 100 tablets; however, the join of TQ1 and TQ2 may require 5000 ranges to be queried; the Batch Scanner can condense these 5000 ranges to a maximum of 100, since there are only 100 tablets, and even find overlapping ranges and merge them. Using the Batch Scanner to process the join, a particular tablet (server in the cluster) is not accessed multiple times for the same query, so the performance is much improved (similar with using clustered indexes versus non-clustered indexes in traditional relational databases).

### 5.3 Time Ranges

All of the examples used up to now in the paper are of data that is mostly static in nature: students, courses, professors, etc. In many applications, data can be very dynamic and have a time component. For example, in a load monitoring application for a data center, the CPU load is measured every 10 seconds and stored in the RDF store; suppose that one is interested in determining the CPU load for the last 30 minutes. In a naive approach, a query may be written as

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

and evaluated like this: 1) Return all cpuLoad measurements, 2) For each measurement, return the timestamp, 3) Perform a client-side filter for measurements that have timestamps > 30 minutes ago. Depending on how much cpu-load data is currently in the store, this query could take a very long time to return. Given a year's data, the first triple pattern (?measurement, cpuLoad, ?load) returns over four million rows, which have to be joined with the timestamp retrieval to return another four million rows. This result will eventually be filtered to return only the last 30 minutes of data, about 180 rows. This naive approach is clearly not feasible in a real world setting. Instead, the timestamp filter can be pushed to the server side in two ways.

First, we suggest using the tablet level scan filters provided by Accumulo. Each table row contains a timestamp (in milliseconds), which by default records when the row was inserted. Accumulo provides an Age Off Filter, which accepts a time to live (TTL) parameter and runs on the tablet level to return only the rows with the insert timestamp greater than the current time minus TTL. Rya provides a mechanism to specify this TTL value with each query submitted. For example, in the above query, the TTL value could be 108000000 (30 minutes in milliseconds). The first triple pattern would then only return the cpuLoad measurements stored in the last 30 minutes. Instead of four million rows returned to perform the join, only 180 rows would be returned. The TTL parameter significantly cuts down on processing time and network utilization of each query. Nevertheless, the row insertion time and the predicate "timestamp" recorded in the RDF triple can be very different, so use of TTLs requires understanding of when the data is measured and when it is inserted into Rya. However, for most queries, any TTL value (even for the last few hours, to be conservative) would be better than retrieving all the data stored.

A second approach to push the timestamp filter down, is to specify a time range in the underlying scan itself. In the above SPARQL query, the line "?measurement timestamp ?ts" will be by default converted to a scan for range [(timestamp,\*,\*), (timestamp,\*,\*)] in the POS table, which will in effect return all the triples in the store with predicate "timestamp". Instead, by defining a custom SPARQL function (timeRange), we can instruct the Rya query planner to build the timestamp range directly into the underlying scan. For example, for the time range [13141201490, 13249201490], the SPARQL query would be modified to:

```
SELECT ?load
WHERE {
  ?measurement cpuLoad ?load.
  ?measurement timestamp ?ts.
  timeRange(?ts, 13141201490, 13249201490).
}
```

The corresponding underlying scan will now be for range [(timestamp,13141201490,\*), (timestamp,13249201490,\*)] in the POS table. Instead of returning all timestamp triples, the scan range would only return timestamp triples that are within the range specified. It is important to note that only time values that are lexicographically sorted can use this approach.

### 5.4 Optimized Joins with Statistics

Another problem with the naive approach to the triple store is the execution order of the query. As described in the examples above, a multi triple pattern query starts processing the first triple pattern and joins the results with each subsequent triple pattern. Depending on the query, this type of processing can seriously affect performance. For example, consider the query "find all students that take Course0". The query can be broken down into two triple patterns: TQ1: (?x, takesCourse, Course0) and TQ2: (?x, type, Student). Only a few (100) triples satisfy TQ1; however, a lot more triples (400K) satisfy TQ2. In this case, the order of the triple patterns works well because the join of TQ1 to TQ2 will only join 100 rows. However, if the query were reordered as: TQ2 followed by TQ1, the process will take significantly longer because the first triple pattern returns 400K rows to be joined with the second triple pattern.

To estimate the selectivity of the queries, we developed a utility that runs MapReduce jobs in Hadoop to count all the distinct subjects, predicates, and objects stored in Rya. These counts are then stored in a table for later retrieval. For example, for the LUBM University dataset, a possible output of this job, in (row, column, value) format used by Accumulo, is given in Table 5:

Row	Column	Value
type	predicate	1 million
student	object	400K
takesCourse	predicate	800K
Course0	object	200

Table 5: Sample University Data set Statistics

The table provides a count for how often a particular value is a subject, predicate, or object. Before a query is run, the selectivity of each triple pattern is estimated based on the values in the statistics table, and execution proceeds using the most selective triple pattern first. In the example, the triple pattern TQ1 (?x, takesCourse, Course0) should take precedence in the join over TQ2 because it limits the data to 200 results (the minimum of the size of Course0 and takesCourse, Min(200, 800K)).

The statistics only need to be updated if the distribution of the data changes significantly because the query planner will reorder the query based on which triple pattern has the highest selectivity. If the data distribution does not change much over time, the query planner will not produce a different ordering of the triple patterns.

## 6. PERFORMANCE EVALUATION

This section describes the experimental setup and the performance evaluation of our RDF TripleStore, Rya, as well as comparison with other existing approaches.

### 6.1 Experiments Set-up

Rya is implemented and tested on Accumulo version 1.3.0. For experiments, we used a cluster consisting of 1 Hadoop NameNode/Secondary NameNode/Job Tracker, 10 Hadoop Data/Task Nodes, 1 Accumulo Master, 10 Accumulo Tablet Servers, 81.86 TB

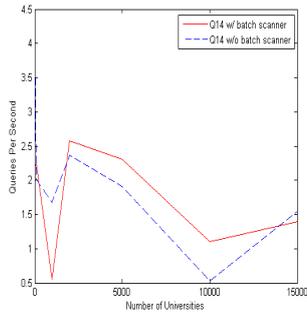


Fig 10. LUBM Query 14

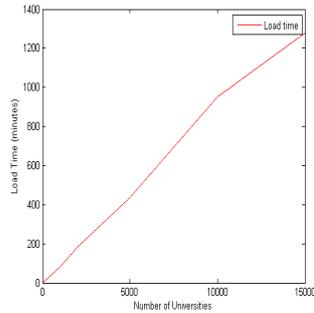


Fig 11. Rya: Data Loading Time

unreplicated storage. Each node has 8 core - Intel Xeon CPU 2.33GHz processor, 16 GB RAM, and 3TB hard drive. We utilized the Lehigh University Benchmark (LUBM) [9]. The LUBM data set is modeled after a set of universities with students, professors, courses, etc.

## 6.2 Performance Results

The main goal of these experiments is to show the scalability of Rya as the size of the data increases. We also want to measure the effect of using the Batch Scanner in improving query performance. We use the *Queries Per Second* metric to report the performance of our system. We measure not only the performance of one query, but of multiple clients interacting with the RDF store simultaneously. In the following tests, we use 8 parallel clients, running a total of 504 runs (24 warmup, 480 measured). A few of the queries return a significantly large result set, which is a measure more of the network performance of transferring a large set of data than of the RDF query performance. To alleviate this, queries 2, 6, 9, and 14 are limited to returning the first 100 results.

**Query Performance:** We generated the following datasets: 10, 100, 1000, 200, 5000, 10000, and 15000 universities. The number of triples varies from 1.3M for 10 universities, to 2.1 billion tuples for 15000 universities. The tests are run using LUBM queries: 1, 2, 3, 4, 5, 6, 7, 8, 9, 14. Queries 10-13 are not currently supported by Rya. The test runs are performed with the performance enhancement, use of the batch scanner, turned on and off. The other performance enhancements, gathering statistics, etc. are necessary; otherwise, many of the benchmarked queries do not return in reasonable time. Query 2 and 9 do not include measurements with the performance turned off because these queries will not return in a reasonable amount of time without all the performance enhancements.

Figure 1 to Figure 10 plot the performance for each query, with and without the use of the Batch Scanner, as the size of the data increases. The information is summarized in Tables 6 and 7 which show the number of queries per second processed by Rya without and respectively with the use of the Batch Scanner.

Figures 1,3,4,5,6,7,8, and 10, show that for all queries with the exception of Queries 2 and 9, the performance does not degrade much even as the number of triples increases thousands of times. Since the triples are indexed, and the data is distributed, even if the data gets larger, the query times do not vary much because each query only interacts with the nodes where the required data resides. The peaks in the graphs when the Batch Scanner is used are most likely attributed to the dataset getting large enough that the data is being distributed well across multiple nodes.

In general, queries benefit from the use of the Batch Scanner. Figure 6 and Figure 10 show that queries 6 and 14 do not benefit

much from the use of the Batch Scanner. In fact, for query 6, the performance is worse when the Batch Scanner is used. The reason is that the Batch Scanned is more useful when there are more joins in the queries, and these queries have no joins. In this case, the overhead of using the Batch Scanner can decrease the performance.

Table 7 shows that Rya performs very well for most queries, with 10s to 100s queries per second even for billion triples stored. Queries 2 and 9 run much slower than the rest of the queries. The reason is the triangular relationship that is being queried for. Query 2 asks for all graduate students that are members of a department that is a sub organization of the university they obtained their undergraduate degree from (the students that continue). In Rya, these triangular relationships are answered in a brute force manner in which the query engine iterates through every graduate student and department to find the ones that have a common university.

Q/#Univ	10	100	1K	2K	5K	10K	15K
Q1	61.41	51.8	67.64	86.26	86.33	92.7	66.3
Q3	14.42	26.78	26.28	21.72	24.32	18.61	20.67
Q4	3.08	10.6	3.35	4.7	6.06	5.72	2.36
Q5	31.83	2.46	2.3	1.87	1.89	1.55	1.05
Q6	34.17	1.2	1.98	2.46	2.31	2.35	2.12
Q7	4.16	10.31	6.08	9.77	9.01	8.17	4.31
Q8	0.11	0.22	0.09	0.07	0.07	0.08	0.05
Q14	4.21	2.02	1.68	2.37	1.91	0.53	1.55

Table 6: LUBM Queries 1-14 - No Batch Scanner (Queries Per Second)

#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

Table 7: LUBM Queries 1-14 - With Batch Scanner (Queries Per Second)

**Load Time:** Figure 11 shows that the load time increases linearly with the size of the data, as expected. Rya's load process utilizes the Bulk Import Map Reduce job provided by Accumulo to speed up the ingest. The ingest process is parallelized across the servers, utilizing all the servers as much as possible.

## 6.3 Performance Comparison with Other Systems

We compare the performance of our system with two other systems: SHARD [15], a horizontally scalable RDF triple store that uses Hadoop Distributed File System for storage, and Graph Partitioning, the system proposed by Huang et al [7], which uses a graph-based vertex partitioning scheme to assign data to nodes in a cluster. SHARD code is open-source and we used the default configurations for the code. For Graph Partitioning, the code received from the authors was "very messy" (quote from the email

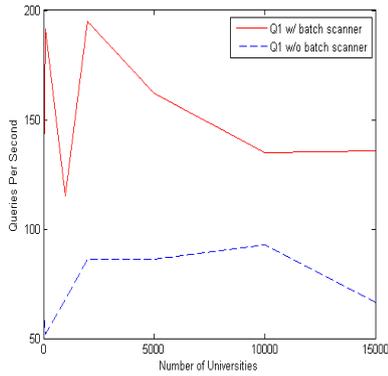


Fig 1. LUBM Query 1

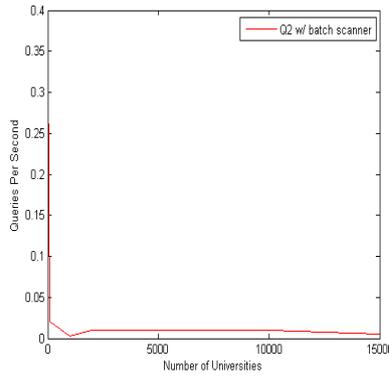


Fig 2. LUBM Query 2

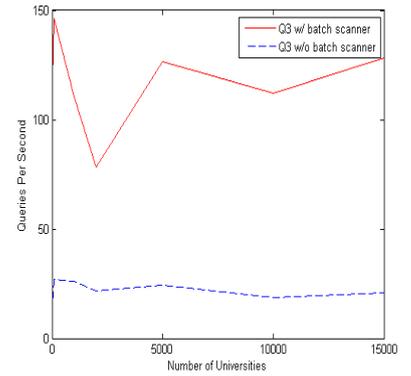


Fig 3. LUBM Query 3

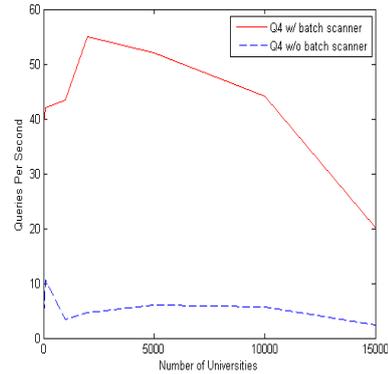


Fig 4. LUBM Query 4

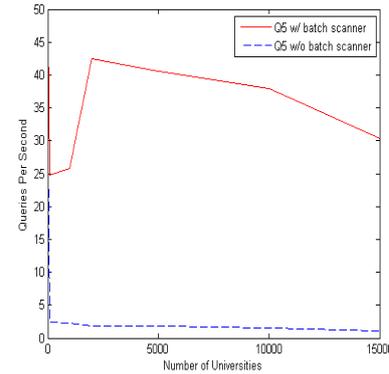


Fig 5. LUBM Query 5

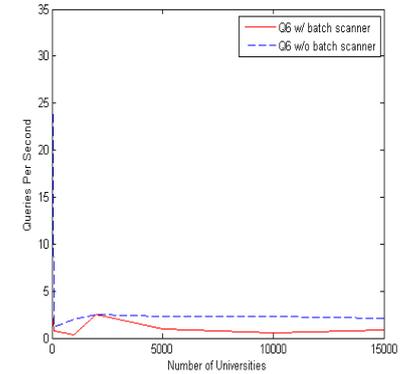


Fig 6. LUBM Query 6

received from one of the authors) and we could not run it on our machines. Therefore we used for comparison the numbers for the "2-hop guarantee" obtained from the authors for the results reported in the VLDB 2011 article [7]. The numbers reported were for experiments run on 20 machines, which leads to better performance than using just 10 machines as in our experiments. However, our goal is to have a rough comparison of our relatively simple system with a more sophisticated system. The data set used was LUBM with 2000 universities, approximately 260 million triples.

Table 8 shows the load time for SHARD, Graph Partitioning, and Rya. Rya is the fastest, since the load process is parallelized and not much data pre-processing is needed to load the three tables.

System	Load Time
SHARD	10h
Graph Partitioning	4h10min
Rya	3h1min

Table 8: LUBM 2000 Data Loading Time

Figure 12 shows the performance results for the LUBM queries supported by all systems under consideration. The performance metric reported is execution time (in seconds) for each query.

For all queries, the performance of Rya and Graph Partitioning is between one and three orders of magnitude better than the performance of SHARD (note the logarithmic scale). The reason for the relatively poor performance of SHARD is that SHARD stores all data in plain files in HDFS, and processing each query involves scanning the entire data set, possibly multiple times if the query contains multiple clauses. Due to the large size of the data, even a parallel scan that uses Hadoop's native Map-Reduce framework

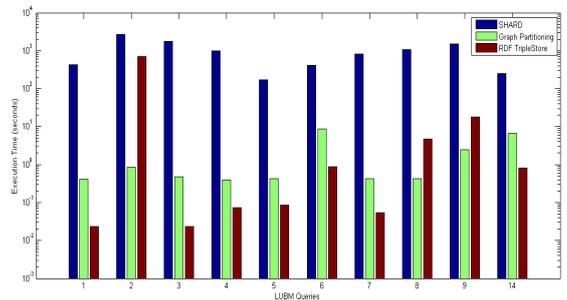


Figure 12: LUBM Execution Time

takes a long time. Rya benefits from the native indexing of tables in Accumulo, and our own three index tables storage solution, so full scans of the data are not performed. Graph Partitioning uses efficient single-node RDF stores which operate in parallel for most of the processing, so unnecessary full data scans are avoided.

Rya is 5-20 times faster than Graph Partitioning for 7 out of the 10 benchmark queries (queries 1, 3, 4, 5, 6, 7, 14). This shows that our indexing scheme and query evaluation methods are competitive, even compared with more complex methods that take advantage of state-of-the-art single-node RDF stores.

For queries 2, 8, and 9, the performance of the Graph Partitioning is better. As explained in Section 6.2, queries 2 and 9 search for triangular relationships, and our approach is rather brute-force in this case; a lot of network traffic is involved in processing the queries and transferring intermediate results. The Graph Partitioning method, with the 2-hop guarantee, is particularly suited for this

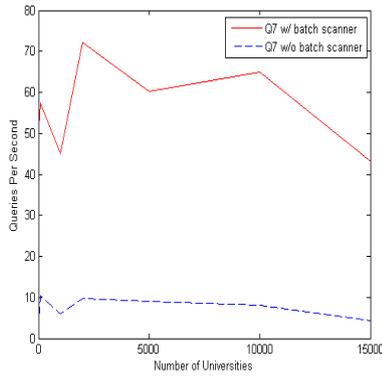


Fig 7. LUBM Query 7

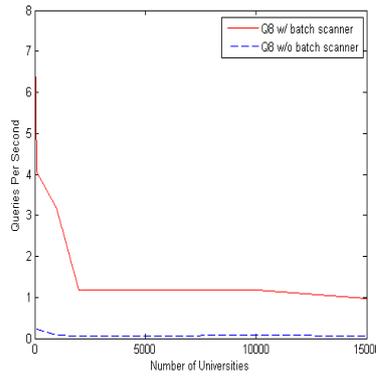


Fig 8. LUBM Query 8

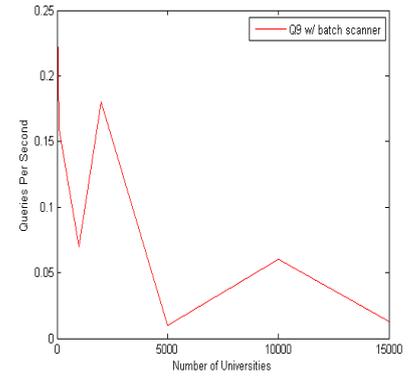


Fig 9. LUBM Query 9

type of queries, as all the data needed is available at one node. Much of the time needed to execute query 8 in Rya involves returning the rather large result set (about 6000 tuples). In Graph Partitioning, the query processing is either done in Hadoop, in which case the results are left in the HDFS, or at the single-node RDF stores, in which case the results are left on the local disk.

## 7. RELATED WORK

There have been several approaches to building an RDF triple index on a Google Bigtable platform. Weiss et al. [18] describe a Hexastore, six indexes on triples to answer the triple patterns. We showed that only three indexes are needed to answer all the triple patterns. Myung et al. [10] describe a SPARQL implementation on top of Hadoop MapReduce. We showed that MapReduce is not necessary at the query level. Starting MapReduce jobs in Hadoop adds unnecessary computational time to each query. A fast SPARQL implementation can be achieved by directly using the Google Bigtable APIs, similar with our use of the Accumulo API. Huang et al. [7] describe an efficient graph partitioning algorithm to separate data across multiple nodes using an arbitrary RDF store at each node with an overarching Hadoop framework to process queries at those nodes. The use of Hadoop MapReduce for some of their queries causes performance lags. We showed in our performance evaluation section that our system outperforms the Graph Partitioning system in many, but not all, cases. Sun et al. [17] describe a system of adapting HBase, to a Hexastore as described in [18], similar to our approach in storing data. However, the mechanism of storing the individual values (e.g. predicate) as the row key, may cause significant problems in scaling due to the fact that HBase has limits on how large a row can be. A predicate such as "rdf:type" could become a very large row and cause problems with the HBase. In addition, the usage of Hadoop MapReduce to query HBase is unnecessary since much of the computation can be done more efficiently by using the HBase APIs directly.

## 8. CONCLUSIONS AND FUTURE WORK

In this paper, we present Rya, a scalable RDF store that uses Accumulo. Based on the storage mechanism implemented in Accumulo, we proposed a three table index for indexing RDF triples. We implemented this RDF store as a plugin to the OpenRDF Sesame framework to give us the ability to accept SPARQL queries and load various RDF formats. We implemented performance enhancements ranging from gathering statistics to using built in Accumulo functions. We have been able to build an RDF store that does basic inferencing, scales to billions of triples, and returns most queries in under a second. For future work, we plan to implement a broader set of inferencing rules, to include owl:sameAs, owl:inverseOf, etc.

We also plan to implement new join algorithms (such as merge join) to possibly speed up some of the slow queries.

## 9. REFERENCES

- [1] Accumulo. <http://wiki.apache.org/incubator/accumuloproposal>.
- [2] AllegroGraph. <http://www.franz.com/agraph/allegrograph/>.
- [3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, and D. A. Wallach. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.
- [4] Hadoop. <http://hadoop.apache.org/>.
- [5] S. Harris and N. Gibbins. 3store: Efficient bulk rdf storage. In *Workshop on Practical and Scalable Semantic Web Systems*, 2003.
- [6] HBase. <http://hbase.apache.org/>.
- [7] J. Huang, D. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. In *VLDB*, 2011.
- [8] D. Kolas, I. Emmons, and M. Dean. Efficient linked-list rdf indexing in parliament. In *Scalable Semantic Web*, 2009.
- [9] LUBM. <http://swat.cse.lehigh.edu/projects/lubm/>.
- [10] J. Myung, J. Yeon, and S.-g. Lee. Sparql basic graph pattern processing with iterative mapreduce. In *2010 Workshop on Massive Data Analytics on the Cloud*, 2010.
- [11] T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. In *VLDB*, 2008.
- [12] OpenRDF. <http://www.openrdf.org/>.
- [13] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. Lopez, G. Gibson, A. Fuchs, and B. Rinaldi. Ycsb++: Benchmarking and performance debugging advanced features in scalable table stores. In *SOC*, 2011.
- [14] RDF. <http://www.w3.org/rdf/>.
- [15] K. Rohloff and R. Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: The shard triple-store. In *International Workshop on Programming Support Innovations for Emerging Distributed Applications*, 2010.
- [16] G. Sanjay, G. Howard, and L. Shun-Tak. The google file system. In *SOSP*, 2003.
- [17] J. Sun and Q. Jin. Scalable rdf store based on hbase and mapreduce. In *Advanced Computer Theory and Engineering, 2010 3rd International Conference*.
- [18] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *VLDB*, 2008.
- [19] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient rdf storage and retrieval in jena2. In *SWDB*, 2003.