SPARQL in the Cloud using Rya

Roshan Punnoose

TexelTek Inc, USA

Adina Crainiceanu

US Naval Academy, USA

David Rapp

Laboratory for Telecommunication Sciences, USA

Abstract

SPARQL is the standard query language for Resource Description Framework (RDF) data. RDF was designed with the initial goal of developing metadata for the Internet. While the number and the size of the generated RDF datasets are continually increasing, 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 Rya, a scalable RDF data management system that efficiently supports SPARQL queries. 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.

Keywords: RDF triple store, SPARQL, distributed, scalable, cloud

1. Introduction

The Resource Description Framework (RDF) [20] 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.

Email addresses: roshanp@gmail.com (Roshan Punnoose), adina@usna.edu (Adina Crainiceanu), rapp@ltsnet.net (David Rapp)

While very efficient, most existing RDF stores [2, 9, 12, 13, 16, 24] 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, thus allowing 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, indexing methods 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, 18] provides. We show through experiments that our system scales RDF storage to billions of records and provides millisecond query times.

This article is an extended version of 'Rya: A Scalable RDF Triple Store for the Clouds' [19] published in The International Workshop on Cloud Intelligence Cloud-I 2012. The system introduced in this article greatly extends the functionality and performance of the system introduced in the Cloud-I 2012 paper. In particular, we introduce an additional inference engine that supports more types of inference (6 instead of 3), we include support for named graphs, we include support for security features such as authorization and visibility, we introduce methods to increase performance of large scans, we provide support for range queries and regular expressions, and we increase data load time performance (up to 5 times faster). We also provide an abstraction of the persistence layer so multiple back-end storage engines can be implemented for Rya.

This paper is structured as follows: Section 2 summarizes the basics of Accumulo, a Google BigTable [5] variant, and OpenRDF, an open source toolkit for implementing RDF triple stores. In Section 3, we describe our basic approach to the problem of scaling an RDF triple store and present the building blocks of our storage and retrieval scheme for RDF. Section 4 describes the approach used to implement SPARQL query processing and Section 5 describes the proposed performance enhancements that provide fast query response times. Section 6 introduces the API for the persistence and query layer abstraction. Section 7 provides experimental results for Rya using a well-known benchmark, and comparisons with existing RDF solutions. We discuss related work in Section 8 and conclusions and future work in Section 9.

2. Background

2.1. Accumulo

Accumulo [1, 18] is an open-source, distributed, column-oriented store modeled after Google's Bigtable [5]. Accumulo provides random, realtime, read/write access to large datasets atop clusters of commodity hardware. Accumulo leverages Apache Hadoop Distributed File System [8], the open source implementation of the Google File System [6]. In addition to Google Bigtable features, Accumulo features automatic load balancing and partitioning, data compression, and fine grained security labels [18].

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 are used to provide efficient lookups of triples in Rya.

Key					Value
Row ID	Column			Timestamp	
	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 [10] 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. This guarantees that clients without specific access rights are not viewing material beyond their access scope. 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 the standard query language for RDF data. 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 [17] 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. Triple Storage

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 are 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 are grouped into the same tablet/server for faster access. In addition, Accumulo divides large groups based on a configuration parameter and moves 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 is (subject, predicate, *), which requires that the persistence layer returns all triples that match the given subject and predicate. Table 2 shows more examples of triple patterns.

Triple Pattern	Expected results
(subject, *, *)	All triples with the matching subject
(*, predicate, object)	All triples with the matching predicate and object
(*, *, *)	All triples
(subject, predicate, object)	One or more particular triples that match all the fields

Table 2: Sample Triple Patterns

We propose to store the RDF triples in the Row ID part of the Accumulo tables (see Section 3.2 for the tables we create). The triple data and associated information are stored as shown in Table 3. The Row ID stores the subject, predicate, and object of the triple, separated by a separator and followed by a byte marker that specifies the type of the object if available, or type string otherwise. The separator used is the Unicode null character *\u0000*, but a comma is shown here for ease of presentation. If the stored triple is part of a named RDF graph, the graph name is stored in the column family field. The column family is empty for the triples in the default graph. Accumulo provides a cell level security access mechanism that is exposed by the Rya API through the visibility field to allow fine grained security control of the data and queries. A timestamp associated with each triple is also recorded (by default the insertion time into the table).

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

Table 3: Rya Key-Value

The storage format for each object in a triple depends on the type information. Rya implements resolvers that specify how to store certain types (string, URI, datetime, double, integer, long, etc). For example, a particular double is stored as 0100123, which translates to 1.23 E100. This allows Rya to perform range queries correctly, based on the actual type of the objects stored, rather than just based on the plain string representation of the objects. Custom types for which Rya does not have a resolver are stored as plain strings.

In Rya, all the data for the triple resides in the Accumulo Row ID. This offers several benefits:

1) by using a 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 and ingest; 3) since no information needs to be stored in the Qualifier, or Value fields of the Accumulo tables, the storage requirements for the triples are significantly reduced. Section 3.2 describes the tables created to store the RDF data and Section 3.3 describes concrete examples of how the triples are stored and queried.

3.2. Three Tables Index

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, as introduced in Section 3.1, 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 4 shows how all eight types of the triple patterns can map to an Accumulo range scan of one of the three tables.

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 4: Triple Patterns Mapped to Table Scans

In many applications, some of the triple patterns will never occur. For example, it is rare to ask a query and not know the predicate, but know the object. Thus, the OSP table may not be necessary; however, for completeness, we implement all the tables.

3.3. Sample Storage and Querying of Triples

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

Table 5 shows an example triple taken from the LUBM [14] dataset. The triple expresses the fact that a particular professor (subject) earned his/her degree (predicate) from a given university (the object, of type URI). Table 6 shows how the triple is stored in the three table indexes SPO, POS, and OSP. As before, for ease 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
http://Univ0/Professor3	urn:degreeFrom	http://Univ2

Table 5: Sample RDF Triple

Table	Stored Triple as Row ID
SPO	http://Univ0/Professor3,urn:degreeFrom,http://Univ2,URI
POS	urn:degreeFrom,http://Univ2,http://Univ0/Professor3,URI
OSP	http://Univ2,http://Univ0/Professor3,urn:degreeFrom,URI

Table 6: 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 in the range (?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: ?x if the row exists, empty set otherwise

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, owl:equivalentProperty, owl:inverseOf, owl:SymmetricProperty, and owl:TransitiveProperty 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, or when the OWL model changes. Stage 1 consists of running MapReduce jobs to iterate through the entire graph of relationships and output the implicit relationships found as explicit RDF triples to be stored into the RDF store. The input for Stage 1 are all the triples with one of the predicate of interest, *rdfs:subClassOf* for example. The output for stage 1 is the transitive closure of the graph represented by the input OWL model.

Algorithms 1 and 2 show sample pseudo-code of the *map* respectively the *reduce* functions. In the map phase in Algorithm 1, each (`s`, "rdfs:subClassOf", `o`) input triple is mapped to two (key, value) outputs. The two output keys are the subject and respectively the object in the triple (`s` and `o` in the example), and the corresponding values represent the original relationship, from the point of view of the key: `s` is the "parentOf" `o`, and `o` is the "childOf" `s`. "parentOf" and "childOf" are just meaningful names we created for the MapReduce jobs. The input to the reduce function in Algorithm 2 is a key and list of corresponding values, both "parentOf" and "childOf" type, produced during map phase. If a given key is the "parentOf" some `x` and "childOf" `y`, then we can infer that `y` is "rdfs:subClassOf" `x`.

Algorithm 1 : `map(triple)`

```
1: {input: triples where predicate = rdfs:subClassOf}
2: //read from input
3: (subject, rdfs:subClassOf, object) triple
4: //divide the triple in two relationships and output (key, value) pairs
5: output (subject, (”parentOf”, object))
6: output (object, (”childOf”, subject))
```

Algorithm 2 : `reduce(key, values)`

```
1: {key: subject or object for some rdfs:subClassOf relationship}
2: {values: list of ”childOf” and ”parentOf” relationships for given key}
3: for each input ”parentOf” relationship (key, (”parentOf”, v1)) do
4:    for each input ”childOf” relationship (key, (”childOf”, v2)) do
5:       //output a triple for inferred relationship
6:       output (v2, rdfs:subClassOf, v1) triple
7:       //increment number of relationships found
8:       increment(SUBCLASSOFRELATIONSHIPS, 1)
9:    end for
10: end for
```

The MapReduce job runs multiple times, until no new relationships can be found. In each iteration, the number of parent/child relationships found is counted. If the same number of relationships is found by two consecutive iterations then we can safely assume that no new relationships can be found (the entire graph has been traversed), and Stage 1 is over. 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”, ”rdfs:subPropertyOf” and ”owl:equivalentProperty” inferences. For example, consider the following SPARQL query:

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

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)
```

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 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. In addition to the query expansion, a marker is added to each StatementPattern expanded, *subclass-expanded*, to identify the nodes that have already been expanded, so query expansion is not applied again. 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)
```

This plan corresponds to an expanded SPARQL query equivalent to the initial query:

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

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}.

The expansion technique is applied to ”rdfs:subPropertyOf” and ”owl:equivalentProperty” inferences as well. In the case of ”rdfs:subPropertyOf”, the query planner searches for every StatementPattern in the query evaluation plan that has a predicate defined (literal value, not a variable) and expands it into a join. Below is an example of how a query containing the predicate ”ub:worksFor” is expanded.

Original SPARQL Query

```
SELECT ?prof
WHERE {
    ?prof ub:worksFor "Department0"}
```

Original Query Evaluation Plan

```
QueryRoot
   Projection
      ProjectionElemList
         ProjectionElem "prof"
      StatementPattern
         Var (name=prof)
         Var (value=ub:worksFor)
         Var (value="Department0")
```

Expanded SPARQL Query

```
SELECT ?prof
WHERE {
    ?pred rdfs:subPropertyOf ub:worksFor.
    ?prof ?pred Department0}
```

Expanded Query Evaluation Plan

```
QueryRoot
   Projection
      ProjectionElemList
         ProjectionElem "prof"
      Join
         StatementPattern
            Var (name=pred)
            Var (value=rdfs:subPropertyOf)
            Var (value=ub:worksFor)
            Var (name=subprop_expanded)
```

```
      StatementPattern
         Var (name=prof)
         Var (name=pred)
         Var (value="Department0")
         Var (name=subprop_expanded)
```

4.3. Using TinkerPop Blueprints

In Stage 1 of the query planner introduced in Section 4.1, we run MapReduce jobs to flatten the ”rdfs:subClassOf”, ”rdfs:subPropertyOf”, and ”owl:equivalentProperty” relationships and store the results in the Rya store. Then, during Stage 2 of the query processing, we use query expansion to add querying the flattened relationships to the query plan. This approach works very well, but in some cases presents a few disadvantages. First, whenever the model changes, the MapReduce job needs to run to update the data. Second, the model is usually quite small, so most of the processing can be done locally, instead of running the MapReduce job to perform the analysis. Third, the query expansion requires a separate Accumulo query for every RDF query. We propose now an extension to the query planner that utilizes graph analysis at the Rya master site. This approach has several advantages: 1) the inferred graph can be cached in memory and graph traversals can be used to find the relationships faster, without querying the Accumulo tables; 2) the graph model can be constructed periodically to obtain the latest inferred graphs; 3) query expansion can be done locally using the in-memory graph model, so a separate Accumulo scan to find the inferences is not needed.

In order to construct the in-memory graph corresponding to an OWL model, we extend the TinkerPop Blueprints [3] implementation of the OpenRDF SAIL interfaces. Once the graph model is constructed, the in-memory graph can be used during query execution phase. For example, consider again the SPARQL query from Section 4.2:

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

By traversing the model graph starting from ”ub:Faculty” node and following backwards ”rdfs:subClassOf” edges, the FullProfessor and Professor nodes are found. The Query Evaluation plan produced by the SAIL API is extended to include the nodes found as follows:

```
QueryRoot
   Projection
      ProjectionElemList
         ProjectionElem "prof"
      Join
         FixedStatementPattern
            Var (name=type)
            Var (value=rdfs:subClassOf)
            Var (value=ub:Faculty)
            Var (name=subclass_expanded)
            Statement (ub:Faculty, rdfs:subCLassOf, ub:Faculty)
```

```
      Statement (FullProfessor, rdfs:subCLassOf, ub:Faculty)
      Statement (Professor, rdfs:subCLassOf, ub:Faculty)
   StatementPattern
      Var (name=prof)
      Var (value=rdf:type)
      Var (name=type)
      Var (name=subclass_expanded)
```

When the query plan is evaluated, the FixedStatementPattern is evaluated locally and results in the "type" variable being bound to "ub:Faculty", "FullProfessor", and respectively "Professor". The second statement pattern is evaluated on the triples stored in the Rya store (in the underlying Accumulo tables). This plan in effect corresponds to the expanded SPARQL query:

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

The rdfs:subPropertyOf, owl:equivalentProperty, owl:inverseOf, owl:SymmetricProperty, and owl:TransitiveProperty inferences are supported in a similar way.

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 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, the optimal setting depends entirely on the processor speed, number of cores, and the type of query.

5.2. Scanner and 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).

The Batch Scanner supports efficient lookup of many ranges, when each range contains small amounts of data. However, using the Scanner is more efficient when few large ranges are requested. The Rya query engine decides at runtime which option to use, and uses the Scanner or the Batch Scanner depending on the individual query.

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 Accumullo. 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.

5.4. Range Queries

Section 3.1 introduced the serialization format for the RDF triples, which includes type information for each object. This information allows us to change how data is stored internally. For example, long integers are padded with zeros to enable correct results to comparison operations in serialized format. This allows Rya to support range queries similar with the time range queries described above, but for more types of data. In particular, Rya supports range queries for objects of type integer, long, double, datetime, boolean, byte, string, and any lexicographically sorted custom data type.

5.5. Regular Expression Filter

SPARQL supports regular expression matching by using the FILTER regex function. Rya supports the regex functionality by incorporating regular expressions for subject, predicate, or object into the Accumulo scans directly. For example, let us assume that one asks for all subjects with age between 10-20 and the name starting with Bob.

```
SELECT ?name
WHERE {
   ?name age ?age.
    FILTER (regex(?name,'^Bob') && ?age >= 10 && ?age <= 20)
}
```

The corresponding underlying scan will be for range [(age, 10, regex:Bob*), (age, 20, regex:-Bob*)] in the POS table. While retrieving the triples in the range [(age, 10, *), (age, 20, *)], the Accumulo-level scan checks the regular expression for the subject *regex:Bob** and returns only the subjects that match the pattern specified by the regular expression. This saves time and network bandwidth, since the processing is done at the Accumulo level, not as a post-processing step.

5.6. Optimized Joins with Statistics

A problem with the naive approach to implementing the triple store is the order in which the triples in a query are evaluated. By default, 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)
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 would 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 7:

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

Table 7: Sample University Dataset 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) takes 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. Persistence Layer Abstraction

There are many different types of NoSQL databases, including Accumulo, HBase [10], Cassandra [4], etc.. While Rya is built on top of Accumulo and takes advantage of some of the Accumulo specific features, such as security features and batch scanner, to some extent, Rya could be built on top of any NoSQL columnar database. To facilitate the use of different backend storage units for Rya, we developed Rya Data Access Object (RyaDAO), a simple abstraction of the persistence and query layer. Instead of having to implement the full SAIL interface for every persistence layer, developers can implement the RyaDAO interface.

The main components of the interface are shown in Table 8. The most basic form of a RyaStatement, represents an RDF triple (subject, predicate, object) or an RDF triple pattern. Other optional fields, shown in italic, are available and can be used, depending on the application needs.

The RyaDAO API includes the following methods: `init()` to initialize the persistence layer (for example to create the underlying tables needed or to connect to the existing tables); `destroy()`, to be used for any clean-up tasks such as flushing and closing any open connections to the database; `add(statement)` to insert an RDF triple into the underlying storage, and `delete(statement)` to delete a triple from the underlying storage.

The RyaQueryEngine API has only one method, `query(statement)`, that takes an RDF triple pattern and returns an iterator to access all the triples in the underlying storage matching the pattern.

RyaStatement	RyaDAO	RyaQueryEngine
subject	init()	Iterator<RyaStatement> query(tripplePattern)
predicate	destroy()	
object	add(statement)	
columnFamily	delete (statement)	
columnQualifer	RyaQueryEngine getQueryEngine()	
columnVisibility		
value		
timestamp		

Table 8: Persistence Layer Abstraction

7. Performance Evaluation

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

7.1. Experiments Set-up

We implemented Rya as described in Sections 3.1, 3.2, 4.1, 4.2, with the performance enhancements described in Section 5. 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 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) [14, 7]. The LUBM data set is modeled after a set of universities with students, professors, courses, etc.

7.2. Performance Results

The main goal of these experiments is to show the scalability of Rya as the size of the data increases. We also 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. The warmup runs are used not to cache the query results, but to initialize the program. In general, Rya does work behind the scenes to initialize inference graphs, thread pools, connections to Accumulo, etc, and the warmup queries give time for Rya to initialize properly before a run is completed. In a real system, Rya would be always running, so the warmup would be already done when a user writes queries. 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 SPARQL 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. Table 9 shows the number of triples in the data sets used for the experiments. For the largest data set used, 15000 universities, there are approximately 2.1 billion triples. 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 enhancements turned off because these queries will not return in a reasonable amount of time without all the performance enhancements.

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

Table 9: Number of Triples

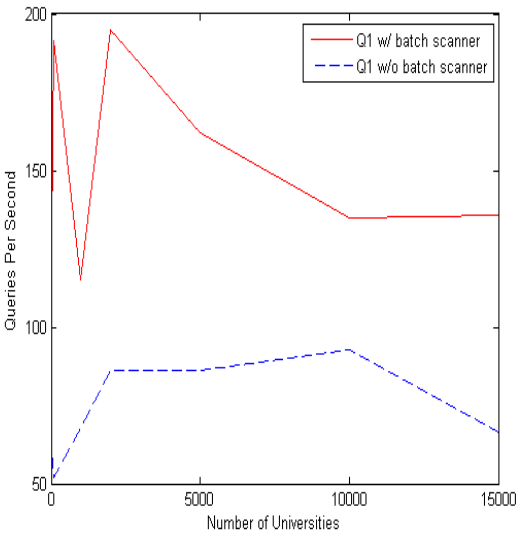

Fig 1. LUBM Query 1

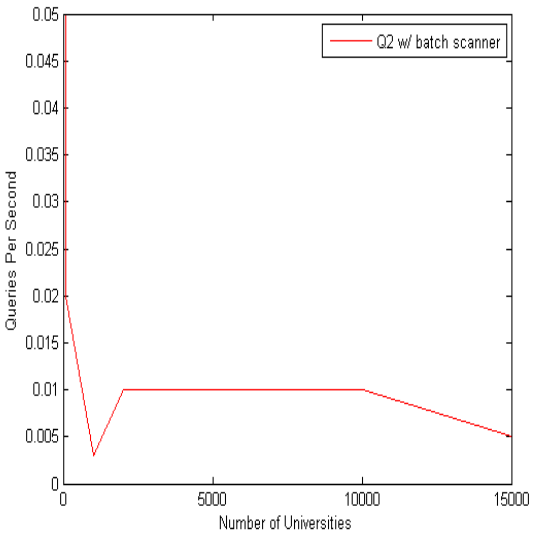

Fig 2. LUBM Query 2

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 10 and 11 which show the number of queries per second processed by Rya without and respectively with the use of the Batch Scanner.

Figures 1 through 10, show that for all queries, 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. The limit for each tablet is set to 256MB. As a tablet grows and reaches this threshold, Accumulo splits it and moves one tablet to a separate node. Most likely at the peak points, the data for some of the tablets reached this threshold and split into multiple tablets. The Accumulo batch scanner is designed to parallelize the data retrieval across multiple tablets. Hence, the more tablets there are, within the number of threads allocated to the scanner, the faster the data can be retrieved.

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 Scanner 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 11 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

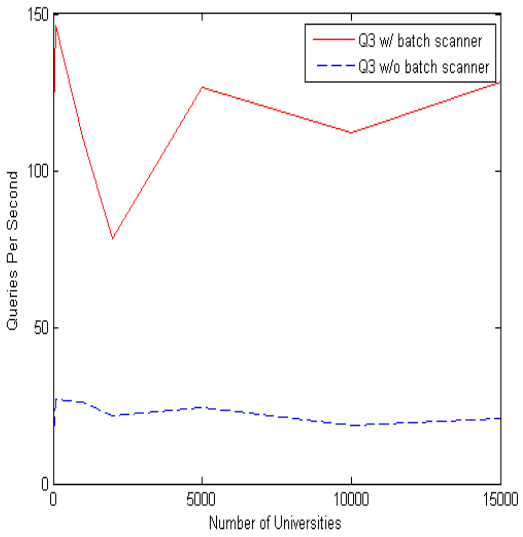

Fig 3. LUBM Query 3

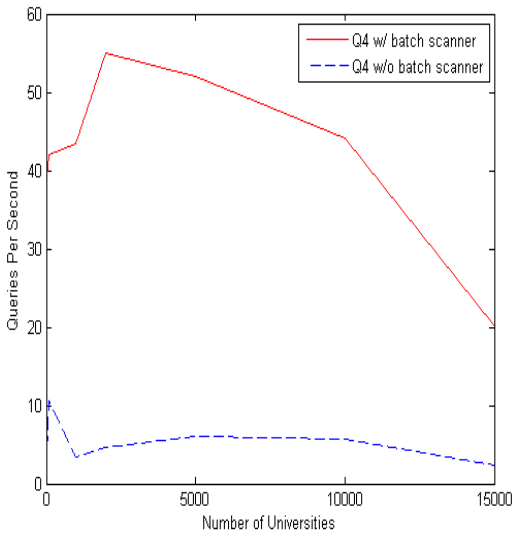

Fig 4. LUBM Query 4

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 10: LUBM Queries 1-14 - No 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 MapReduce 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. By using compression in the MapReduce job, the loading time can be decreased further.

7.3. Performance Comparison with Other Systems

We compare the performance of our system with three other systems: SHARD [21], a horizontally scalable RDF triple store that uses Hadoop Distributed File System for storage, Graph Partitioning, the system proposed by Huang et al [11], which uses a graph-based vertex partitioning scheme to assign data to nodes in a cluster, and RDF-3X [16], a state-of-art single-node

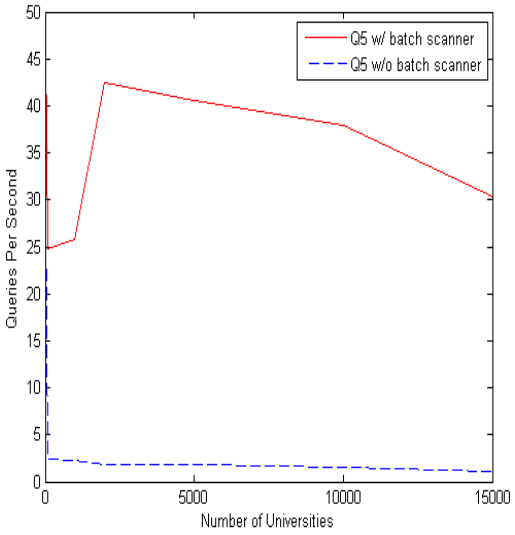

Fig 5. LUBM Query 5

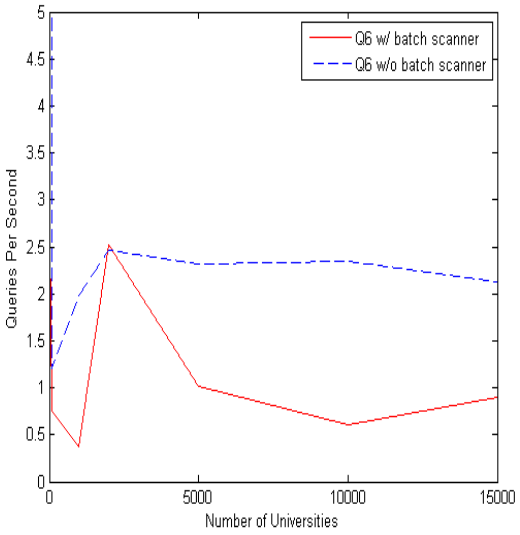

Fig 6. LUBM Query 6

triple store. 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 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 [11]. 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. We also compared with RDF-3x, to see whether it would be faster to use a traditional triple store, if it can store the data. The data set used was LUBM with 2000 universities, approximately 260 million triples. For these experiments, we run Rya with no limit on the size of the result set, except for queries 2 and 9 which are limited to 100 because the size of the intermediate results is very large and the queries did not return in reasonable time without the limit.

Table 12 shows the load time for SHARD, Graph Partitioning, RDF-3X, and Rya. The RDF-3X is the fastest, likely because less communication overhead is required. Among the distributed systems, Rya is the fastest, since the load process is parallelized and not much data pre-processing is needed to load the three tables.

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 up to 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 takes a long time. Rya benefits from the native index-

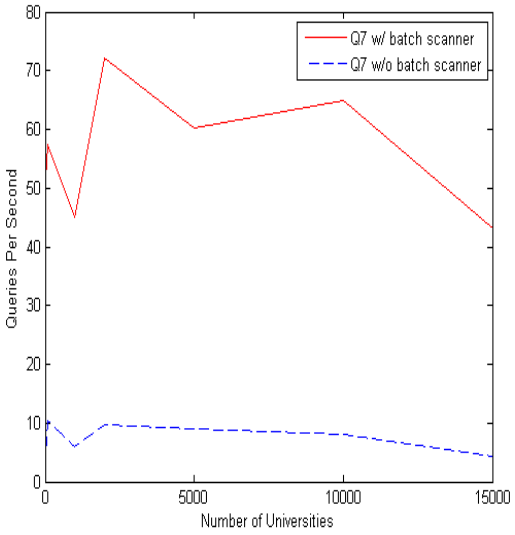

Fig 7. LUBM Query 7

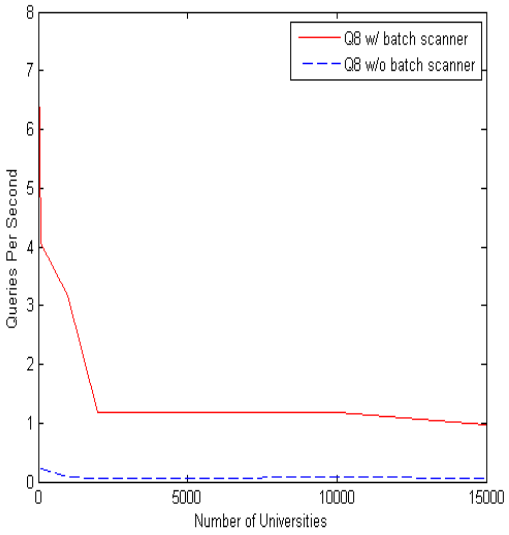

Fig 8. LUBM Query 8

ing 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. For queries 6 and 14, SHARD's performance is close to Rya's. Those queries have large input, low selectivity, and involve no joins. Query 6 does involve implicit and explicit "subClassOf" relationships, but SHARD pre-computes all such relationships and stores them in the triple store, so only a scan is needed to answer the query. The cost for Rya comes from the network traffic required to return the very large result set. If the result set is limited to 100 tuples, Rya's performance is much improved (0.87 seconds with the limit vs. 358 seconds without the limit for query 6 and 0.81 seconds with the limit vs. 303 seconds without the limit).

Rya is 5-20 times faster than Graph Partitioning for 5 out of the 10 benchmark queries (queries 1, 3, 4, 5, 7). 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, 6, 8, 9, and 14, the performance of the Graph Partitioning is better. As explained in Section 7.2, 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 type of queries, as all the data needed is available at one node. Much of the time needed to execute queries 6, 8, and 14 in Rya involves returning the rather large result set (about 6000 tuples for query 8). 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. If we limit the result set to 100 for queries 6 and 14, Rya performs faster than Graph Partitioning.

Rya's performance is better than RDF-3X for 5 queries, and slightly worse for the queries that either involve triangular relationships, or have large results set, both cases involving large

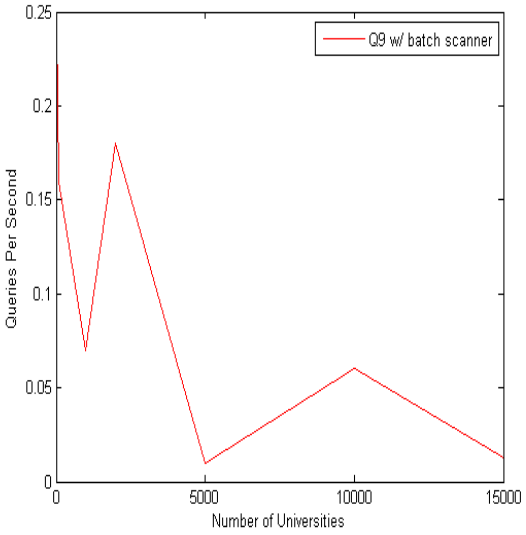

Fig 9. LUBM Query 9

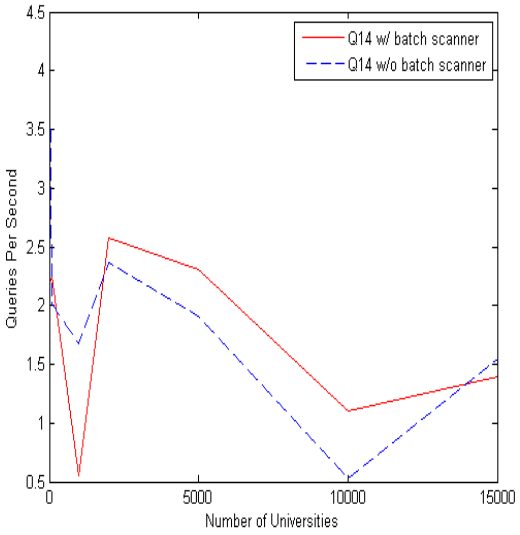

Fig 10. LUBM Query 14

amounts of data being transferred over the network. Given that RDF-3X is a state-of-art single node triple-store, highly optimized, these results show that Rya is highly competitive. Moreover, Rya is able to scale to data sizes impossible to handle in a single-node triple store.

8. Related Work

There has been extensive work in creating efficient RDF triple-stores in a centralized setting [2, 9, 12, 13, 16, 24]. While these systems are very efficient, they were not created for a distributed environment. With the massive increase in data size, solutions that scale in a distributed environment are needed.

There have been several approaches to supporting SPARQL queries using MapReduce and/or building RDF triple stores on a Google Bigtable platform. A summary of these approaches is shown in Table 13.

Myung et al. [15] describe a SPARQL implementation on top of Hadoop MapReduce. The data is stored in N-Triples format in the Hadoop Distributed File System, and the query engine focuses on RDF graph pattern matching by implementing selection and join algorithms in the MapReduce framework. OWL inference is not supported by the system. Rya offers support for six types of inferences (rdfs:subClassOf, rdfs:subPropertyOf, owl:equivalentProperty, owl:inverseOf, owl:SymmetricProperty, and owl:TransitiveProperty). By storing the data in Accumulo and using the three indexes, query performance is improved tremendously for many queries, since not all the data needs to be accessed. Moreover, 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 API, similar with our use of the Accumulo API.

Weiss et al. [23] describe Hexastore, a centralized system, in which six indexes corresponding to all possible orderings of the elements in a triple are created. Each instance of an RDF element,

#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 11: LUBM Queries 1-14 - With Batch Scanner (Queries Per Second)

System	Load Time
SHARD	10h
Graph Partitioning	4h10min
RDF-3X	2h30min
Rya	3h1min

Table 12: LUBM 2000 Data Loading Time

considered the ”primary” element (subject, predicate, or object), is associated with two sorted ”secondary” vectors, one for each of the other elements in the RDF triple, and each element in the secondary vector is linked to a sorted vector containing the ”third” type of element in the triple. For example, a ”primary” subject s, is associated with two sorted ”secondary” vectors: one sorted vector of all the predicates associated with s, and one sorted vector of all objects associated with s. Each of the elements in the ”secondary” vectors, for example p_i in the predicates vector, is also linked a third vector made of all the object instances corresponding to the given subject s and predicate value p_i. Sun et at. [22] describe a system of implementing the Hexastore on top of HBase, 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. We also showed that only three indexes are needed to answer all the triple patterns. 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 API directly.

Rohloff et. al. [21] introduce SHARD, a horizontally scalable RDF triple store that uses Hadoop Distributed File System (HDFS) for storage. RDF data is aggregated by subject, and stored in flat files in HDFS. Each line in the file represents all the triples that have the same subject. Scanning the data can be done in parallel in the MapReduce framework, and joins on subject are very efficient, as such joins are pre-computed. The query engine iteratively processes each of the clauses in a SPARQL query, attempting to bind query variables to literals, while satisfying all the query constraints. Each iteration involves a MapReduce step which accesses all the data stored in the triple-store plus the intermediate data produced by the previous iterations. Rya

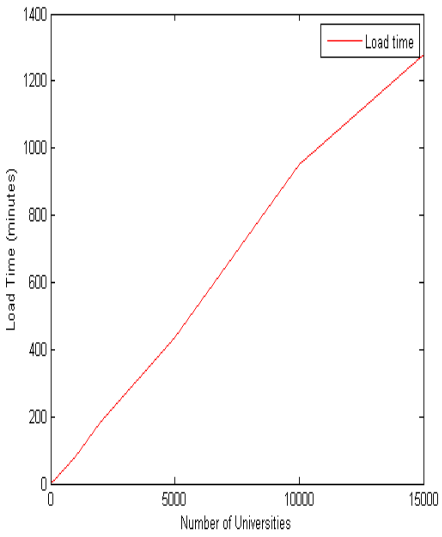

Fig 11. Rya: Data Loading Time

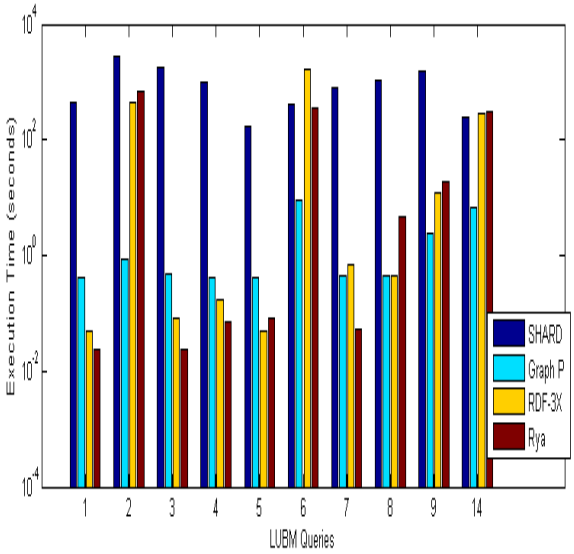

Fig 12. LUBM Execution Time

System	Storage	Query Processing	OWL Inference
[15]	N-Triples in HDFS	MapReduce	No
[22]	6 index tables in HBase	MapReduce	No
[21]	subject-aggregated files in HDFS	MapReduce	No
[11]	single-node RDF stores	RDF stores and/or MapReduce	Yes
Rya	3 index tables in Accumulo	Accumulo API + cost based optimizations	Partial

Table 13: Systems with SPARQL Support Using MapReduce

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. The experimental results in Section 7.3 show that Rya outperforms SHARD, sometimes by as much as three orders of magnitude.

Huang et al. [11] introduce a scalable system for processing SPARQL queries. A graph-partitioning algorithm is used to partition the RDF graph data. Each partition is then stored in a single-node RDF triple store. The query evaluation engine attempts to decompose the query into parallelizable chunks and distribute their evaluation to the individual stores. If a completely parallel execution is not possible, the Hadoop MapReduce framework is used in addition to the single-node triple stores to evaluate the query. Parallel evaluation of the queries in single-node state-of-the art triple stores is very efficient, but the use of Hadoop MapReduce for some of the queries causes performance lags. The performance evaluation results in Section 7.3 show that Rya's performance is comparable, and in many cases superior, to Graph Partitioning.

9. 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 on integrating the merge join and hash join algorithms we implemented, into the Rya query planner. With that, based on the statistics we collect, Rya can select the type of join to perform at any step in the query processing. We started working implementing the RyaDAO interface introduced in this article to use HBase as the backend storage for Rya. Future work includes implementing some of the performance enhancement techniques for HBase, and evaluating the performance of the HBase-based Rya with the Accumulo-based Rya. We also plan on extending Rya to support a broader set of inferencing rules.

10. Acknowledgements

We would like to thank Andrew Skene, Sean Monaghan, and Corey Nolet for their help in developing Rya.

References

[1] Accumulo. http://wiki.apache.org/incubator/accumuloproposal.
[2] AllegroGraph. http://www.franz.com/agraph/allegrograph/.
[3] T. Blueprints. https://github.com/tinkerpop/blueprints/wiki.
[4] A. Cassandra. http://cassandra.apache.org/.
[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 205–218, 2006.
[6] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, 2003.
[7] Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics*, 3(2-3):158–182, Oct. 2005.
[8] Hadoop. http://hadoop.apache.org/.
[9] S. Harris and N. Gibbins. 3store: Efficient bulk rdf storage. In *Workshop on Practical and Scalable Semantic Web Systems*, 2003.
[10] HBase. http://hbase.apache.org/.
[11] J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *Proc. VLDB Endow.*, 4(11):1123–1134, 2011.
[12] D. Kolas, I. Emmons, and M. Dean. Efficient linked-list rdf indexing in parliament. In *Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems*, SSWS2009, pages 17–34, 2009.
[13] X. Liu, C. Thomsen, and T. B. Pedersen. 3xl: Supporting efficient operations on very large owl lite triple-stores. *Information Systems*, 36(4):765–781, June 2011.
[14] LUBM. http://swat.cse.lehigh.edu/projects/lubm/.
[15] J. Myung, J. Yeon, and S.-g. Lee. Sparql basic graph pattern processing with iterative mapreduce. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*, MDAC '10, pages 6:1–6:6, 2010.
[16] T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. *Proc. VLDB Endow.*, 1(1):647–659, Aug. 2008.
[17] OpenRDF. http://www.openrdf.org/.
[18] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi. Ycsb++: benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 9:1–9:14, 2011.

[19] R. Punnoose, A. Crainiceanu, and D. Rapp. Rya: a scalable rdf triple store for the clouds. In *Proceedings of the 1st International Workshop on Cloud Intelligence*, Cloud-I '12, pages 4:1–4:8, 2012.

[20] RDF. http://www.w3.org/rdf/.

[21] K. Rohloff and R. E. Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: the shard triple-store. In *Programming Support Innovations for Emerging Distributed Applications*, PSI EtA '10, pages 4:1–4:5, 2010.

[22] J. Sun and Q. Jin. Scalable rdf store based on hbase and mapreduce. In *Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference on*, volume 1, pages V1–633–V1–636, 2010.

[23] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proc. VLDB Endow.*, 1(1):1008–1019, Aug. 2008.

[24] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient rdf storage and retrieval in jena2. In *The first International Workshop on Semantic Web and Databases*, SWDB, pages 131–150, 2003.