Acacia-RDF: An X10-based Scalable Distributed RDF Graph Database Engine

Miyuru Dayarathna  
WSO2 Inc.  
/ JST CREST  
miyurud@wso2.com

Isuru Herath, Yasima Dewmini  
Gayan Mettananda, Sameera Nandasiri  
Sanath Jayasena  
Dept of Computer Science & Engineering,  
University of Moratuwa  
(isuruherath.11, yasima.11, gayan.11,sameera.11,sanath)@cse.mrt.ac.lk

Toyotaro Suzumura  
IBM T.J. Watson Research Center  
/ JST CREST  
suzumura@acm.org

Abstract—Linked data mining has become one of the key questions in HPC graph mining in recent years. However, the existing RDF database engines are not scalable and are less reliable in heterogeneous clouds. In this paper we describe the design and implementation of Acacia-RDF which is a scalable distributed RDF graph database engine developed with X10 programming language to solve this issue. Acacia-RDF partitions the RDF data sets into subgraphs following vertex cut paradigm. The partitioned data sets are persisted on secondary storage across X10 places. We developed a scalable SPARQL processor for Acacia-RDF which operates on top of partitioned RDF data. Furthermore, we design and implement a replication based fault tolerance mechanism for Acacia-RDF. We present performance results gathered from Acacia with different scales of LUBM RDF benchmark data sets and make a comparison of Acacia’s performance against Neo4j graph database server. From the scalability experiments conducted upto 16 X10 places, we observed that Acacia-RDF scales well with LUBM data sets. Acacia-RDF reported less than ten seconds elapsed times on 16 places for running the first and third queries of the LUBM benchmark on LUBM 160 universities data set with 3.6 million vertices and 28.5 million edges which was 1.7GB in size. Through this work we describe and demonstrate the use of X10 language for development of scalable graph RDF data management systems.

Index Terms—RDF; SPARQL; Graph databases; Database management; Distributed databases; Graph theory; Cloud computing;

I. INTRODUCTION

Data in the form of linked/graph data have become prominent in recent computing applications. Examples for such applications are spread across multiple applications such as online social networks, semantic web (DBPedia), major search engines (e.g., Google, Yahoo!, Bing). Facebook’s Like button, BBC’s wildlife and music pages are some examples for use of linked data [24]. Linked data provides a set of techniques for interacting with structured data on the web. Resource Description Framework (RDF) is a standard model for data interchange on the Web which supports this interaction [23]. A single RDF statement describes a relationship between two entities. These three elements are called subject, predicate, and object in the Linked Data terminology and is often referred to as a triple. RDF is the data model used by semantic-web ontologies and knowledge bases such as DBPedia, Probase, YAGO, etc. Number of database systems have been developed in recent years by both academia and industry to cater the need of managing and mining large linked datasets. Some of the notable examples include Trinity [28], GraphChiDB, AllegroGraph, Titan [2], etc.

Although number of RDF storage systems have been proposed in recent literature they are not specifically intended to run in large scale distributed memory computer systems. Furthermore, many of the current systems do not concentrate on efficient storage of RDF data sets in secondary storage. There are currently few disk-based distributed storage for graphs available.

X10 is a programming language which follows Asynchronous Partitioned Global Address Space (APGAS) model [14][3]. In this paper we describe the architecture of an X10 based distributed RDF graph database system which solves the aforementioned issues. We implement our proposed RDF graph storage system on top of Acacia\(^1\) open source distributed graph database server [8]. RDF graphs in Acacia-RDF are partitioned using Metis graph partitioner [15] and are distributed across the X10 places (an X10 place corresponds to a Java Virtual Machine) in such a way that it reduces the communication overhead during the query processing.

With Acacia-RDF we have done significant architecture change to Acacia [8]. First, we introduced our own native store which eliminated the dependency with Neo4j. Second, we have enhanced Acacia to run not only on clusters but also on single computers such as laptops. Third, we have implemented a scalable SPARQL query processor in X10. Fourth, we implement a replication based fault tolerance mechanism for Acacia-RDF. Finally, we have implemented several additional graph algorithms in Acacia’s distributed data abstraction. However, in this paper our main focus is on describing the general architecture of Acacia-RDF and its performance. In this paper we describe experiments conducted on Acacia-RDF with loading LUBM (Lehigh University Benchmark) [10] data sets of different scales. We have conducted scalability and performance comparison experiments using the first three

\(^1\)https://github.com/miyurud/Acacia
queries of LUBM RDF benchmark. From these experiments we observed scalability of Acacia-RDF by running in a cluster of four computers up to sixteen X10 places (an X10 place corresponds to a Java Virtual Machine) with the first and the third LUBM queries running with 16 places on LUBM 160 data set of 1.7GB with elapsed times of less than 10 seconds. With Acacia-RDF we have developed a first such implementation of a distributed, scalable SPARQL processor completely written in X10. Specifically the contributions of this paper can be listed as follows,

- **Distributed RDF graph database engine** - We describe the architecture of a distributed, scalable RDF graph database engine which partitions and persists RDF graphs across multiple workers.
- **X10-based SPARQL Executor** - We design and implement a SPARQL query processor completely in X10.
- **X10-based Fault Tolerance** - We leverage X10 language’s support for developing fault tolerant graph database server which tolerates the scenarios of worker failures.
- **Evaluation** - We provide performance evaluation results of the Acacia-RDF system with experiments conducted using LUBM data sets up to LUBM scale 160. Furthermore, we demonstrate the scalability of Acacia-RDF.

The paper is organized as follows. We provide related work in Section II, and describe the system design of Acacia-RDF in Section III. Next, we provide the implementation details of Acacia-RDF in Section IV. We describe the evaluation of Acacia-RDF in Section V. We discuss the results in Section VI, and conclude in Section VII.

II. RELATED WORK

Linked data management has been a broadly studied research problem in recent times. Multiple work have been done on implementation of distributed, scalable graph databases. Furthermore, several large scale RDF stores has been described in recent research. Note that the details given below only correspond to the work conducted on RDF/Graph storage and data management and we do not survey graph library/framework related work.

There are few notable distributed graph databases present in the state-of-the-art. G∗ is a distributed graph database server which manages collections of large graphs by distributing storage across multiple servers [17]. System G is another graph database server which provides a whole spectrum solution for RDF storage, runtime, analytics, and visualization [26]. However, none of them have investigated on RDF graph storage. Titan is a distributed graph database server. However, it does not have a native storage as Acacia-RDF. Instead it uses a third party key-value store such as Cassandra, HBase, BerkeleyDB, etc. as its backend storage. SW-Store is an extension of a column-oriented DBMS, that is designed for high performance RDF data management [1]. GoFFish is a software framework for storing graphs and conducting graph analytics similar to Acacia [21]. Different from GoFFish, Acacia-RDF is a distributed RDF graph database engine while GoFFish is yet to support storage and processing of RDF data. Furthermore, GoFFish is implemented using Java where as Acacia-RDF is developed in X10 which runs on top of Managed X10 runtime. Li-Yung Ho et al. [13] described a distributed graph database architecture for large social computing. But their focus was not on RDF graphs. Their underlying communication has been implemented using MPI while Acacia-RDF uses the socket back-end of Managed X10/Java sockets for communication.

DREAM is a distributed RDF engine with adaptive query planner and minimal communication [12]. Different from Acacia-RDF DREAM does not partition RDF data sets. DREAM partitions only SPARQL queries. Although this eliminates the requirement of joining partitioned subgraphs during certain query execution, duplication of the same RDF data set across multiple computers consumes significant storage resources which is different from the objectives of Acacia-RDF. TriAD is a distributed RDF engine which is based on shared-nothing, main-memory architecture which uses an asynchronous message passing protocol [11]. However, in Acacia-RDF communication is done based on Master-Worker pattern.

Zeng et al. implemented Trinity.RDF, a distributed, memory-based graph engine for web scale RDF data [28]. Similar to Acacia-RDF which is built on top of Acacia, Trinity.RDF is based on Trinity graph engine. However, Trinity is a distributed in-memory key-value store while Acacia-RDF persists the graph data sets across multiple compute nodes and systematically loads data into memory during query processing. g-Store is a graph-based RDF data management system which maintains the data as a directed multilabeled graph where each vertex corresponds to a subject or object [30]. However, g-Store is a non-distributed system. SemStore is a semantic-preserving distributed triple store [25]. Different from Acacia-RDF which follows vertex-cut paradigm, SemStore adopt a coarse-grained unit named Rooted Sub-Graph. TripleBit is a compact system for storing and accessing RDF data [27]. Papailiou et al. presented a system that addresses graph-based, workload-adaptive indexing of large RDF graphs by caching SPARQL query results [20]. Zou et al. proposed a semantic query graph to model the query intention in the natural language question in a structural way [29]. RDF Triple Filtering (R3F) is a method that exploits the graph-structural information of RDF data [16]. These are potential optimizations that we may follow in future with Acacia-RDF. RDF-3X engine is pursuing a RISC-style architecture with streamlined indexing and query processing [18]. However, RDF-3X is a non-distributed system.

Graph partitioning is a challenging issue that needs to be addressed in distributed graph data management. We utilize METIS [15] graph partitioner to reduce the communication overhead between compute nodes.

X10 language has been investigated for graph processing previously. ScaleGraph is one of the earliest examples [5]. Furthermore, graph database benchmarks such as XGDBench [7][6] has been developed previously which are aimed for running in distributed clusters. However, Acacia-RDF is the
first such attempt in use of X10 language for storage and processing of RDF data.

### III. System Design

An overview of Acacia-RDF’s system architecture is shown in Figure 1. There are two key components of Acacia system: Master and Worker. Front-end is a command-line user interface for Acacia’s Master. There are Front-end commands to list the system statistics, to upload/delete graphs, to get system statistics, commands to run graph algorithms, etc.

Once a RDF graph is submitted to the system, it extracts the vertices, edges, and their properties. The RDF graph is partitioned using Metis graph partitioner. Each partitioned subgraph is stored in the native graph store structures and the native stores are distributed across X10 places.

During the query execution, SPARQL query submitted by the user is parsed to identify triple patterns present within the query. The identified patterns are matched with the partitioned RDF graph data sets which are located in multiple X10 places. In order to optimize the query execution on Acacia-RDF we have introduced a query results caching mechanism which operates on each and every worker of Acacia-RDF. The query caching mechanism checks to see if the query, the target graph (unmodified) are the same for a previous query execution session. If these parameters are same, then already cached results are sent to the master for aggregation rather than reloading the relevant subgraphs from the disk storage and executing the queries.

There are two main types of SPARQL query executions happen in Acacia-RDF. These are **single variable queries** and **multi-variable queries**. In the case of single variable queries we merge the intermediate results at the workers and send the final results to the master. However, in the case of multi-variable queries we send the intermediate results from each and every worker to the master and merging of the intermediate results is done at the master. This type of intermediate results aggregation has been followed to ensure the correctness of the results obtained from the SPARQL processor.

### IV. Implementation

We implemented Acacia using X10 programming language for which we provide a necessarily brief introduction next.

X10 is an open source programming language that is aimed for providing a robust programming model that can withstand the architectural challenges posed by multi-core systems, hardware accelerators, clusters, and supercomputers [14][3]. The main role of X10 is to simplify the programming model in a way that it leads to increase in programmer productivity for future systems such as Extreme-scale computing systems [9]. X10 is a strongly typed, object-oriented language which emphasizes static type-checking and static expression of program invariants. The latest major release of X10 is X10 2.5 of which the applications are developed by source-to-source compilation to either C++ (Native X10) or Java (Managed X10). We used managed X10 when developing Acacia-RDF. In Managed X10 the X10 application gets translated in to a pure Java application. We leverage the notion of places, language constructs for asynchronous execution (i.e., `async`, `finish`, etc.) available in X10 when developing the Acacia system. Furthermore, we leverage parts of the the built in fault tolerance mechanism of X10 when formulating Acacia’s fault tolerance mechanism [4].

We have made several significant architectural changes with RDF extension for the initial Acacia system described in [8]. The most notable change is the elimination of the Neo4j instances and replacing them with a Native Store developed by us. Furthermore, we had observed that considerable amount of edges stored for certain graphs in central store. With Acacia-RDF we have eliminated this bottleneck by distributing the central store across workers as an when required. Figure 2 shows how this is being done. We follow a random partitioning technique to equally divide the number of edges across central stores located on each instance. Furthermore, the previous version of Acacia’s data loading phase depended on a sequence of MapReduce jobs. With Acacia-RDF we have eliminated this dependency and have introduced a `MetisPartitioner` which constructs METIS file format and conducts graph data partitioning. Therefore, the latest modifications allows the Acacia system to be run even in single computer which allows the system to be used in multiple use cases compared to the previous system. Next, we describe how Acacia’s RDF extension has been implemented on top of Acacia system.

#### A. RDF Data Partitioner and Native Store

Another view of system architecture of Acacia-RDF is shown in Figure 2. Once an RDF data set is submitted to Acacia-RDF it extracts vertices, edges, and their attributes. One of the main challenges in building a scalable RDF engine is how to partition the RDF data across a compute cluster in such a way that queries can be evaluated with minimum communication cost incurred by distributed joins [25]. In order to achieve this goal We use METIS graph partitioner [15] to implement the graph partitioning functionality of Acacia-RDF. The list of edges are partitioned by Metis and the partitioned edges are separated to local stores if the two vertices belongs to
the same subgraph. If not the edge is stored in a central store. Vertex attributes, predicates, and other meta information such as partition ID are stored in separate files within the native storage.

The data structures used within the native store are shown in Figure 3. LocalSubGraphMap stores the edge list of the graph being stored. VertexPropertyMap stores the properties of the vertices. RelationshipMapWithProperties stores relationship information while PredicateStore keeps the predicates. There are several important variables such as IsCentralStore, VertexCount, EdgeCount, PredicateCount, PartitionCount, etc. All these data structures are serialized using Kryo library when storeGraph() method of the native store is called. The stored data is loaded in to memory when loadGraph() is called. Since only objects can be serialized in Java, we transfer variables to a MetaInfo map when serialized and extract those values from the map when deserialized.

Acacia-RDF maintains all the operational information in a meta data store implemented using HSQLDB. Meta information includes details such as ids of graphs, hosts, IP addresses configured with Acacia, partition IDs, sizes of partitions, locations where they are stored, etc.

**B. SPARQL Query Processor**

SPARQL query processor executes SPARQL queries specified by the users on the partitioned RDF graphs of Acacia-RDF. User can either select to list maximum 100 lines of results on the front-end command line interface or store the entire results of query execution on a file. We use an ANTLR-based SPARQL grammar for parsing SPARQL queries [22]. The user specified SPARQL query is transferred to each worker by the Acacia Manager. Each worker runs the SPARQL query they received in parallel and returns back the result to Master based on the above mentioned criteria. Since the RDF graphs are partitioned and stored across multiple places the central stores are consulted when running SPARQL queries which require joining multiple subjects and objects. This is a common architectural feature present across all the graph algorithms implemented in Acacia.

The pseudo codes shown in Algorithms 1, 2, and 3 described the SPARQL executor of Acacia-RDF. Algorithm 1 is the main algorithm which describes how the SPARQL query execution happens while two of its functions executeTriplePattern() and MergeAnswer() are located in Algorithms 2 and 3 respectively. The algorithm 2, is used to identify and process different types of queries. Algorithm 3 is used to merge the intermediate answers and get the final result.
We have drawn a block diagram to describe the main components of Acacia-RDF's SPARQL processor see Figure 4. The SPARQL executor consists of many components. First one is the Executor. It initiates the query processing. Next one is the Tokenizer. It breaks the query into tokens. TriplePattern represents different triple patterns. Triple represents the components of a triple in a query.

During the query execution the Executor initiates the query processing. Tokenizer breaks the query into tokens. Then if the query type is SELECT, then the Executor should return the matching result set to the user. After tokenization, data should be loaded. Data is loaded from Native Store. Next, each triple in the query, should be matched with loaded data in order to find matching results for the triple. After that intermediate results of each triple should be joined to get the final result set. Finally, Executor will return the matching result set for the given query.

Next, we describe how fault tolerance has been implemented in Acacia-RDF.

C. Fault Tolerant Execution

Fault tolerant operation is a key requirement expected from any data management system software because they get deployed in environments with frequent system failures. In this section we describe how fault tolerant operation of Acacia has been implemented.

Crash faults such as operating system halts, power outages, VM crashes, etc. hampers the Acacia's ability to produce correct results of a user query. To avoid this issue we have created a fault tolerant query execution model for Acacia-RDF. Note that in the current model we assume that the environment is fault free during the data uploading phase.

We divide the fault tolerance model of Acacia-RDF into two categories based on the location where the fault occurs. First, the Place 0 is alive, but some other place(atleast one) are dead. Second, the Place 0 is dead and other places (atleast one) are alive.

In the fist scenario once place(s) crash we identify that event by calling isDead() method and identify the node ID by iterating the list of places which were created originally. Then we distribute the assigned activities to the failed node across live nodes. The data for the failed node will be loaded from the replications and distribute among live nodes according to activity distribution.

In this version of Acacia-RDF we have completed implementing the replication distribution algorithm and the first sce-
The experiments conducted on Acacia-RDF are of three folds. In all these experiments we setup Acacia-RDF in a cluster of four computers. The systems were running on Ubuntu Linux, X10 2.5.2, and JDK 1.7.

In the first experiment, Acacia-RDF was configured to run with max 8GB heap, 4 places. The aim of the experiment was to compare performance of Acacia-RDF with Neo4j graph database server. We used four LUBM data sets of the sizes listed in Table I during the experiments. We used Neo4j 2.2.4 in this experiment and batch uploaded the LUBM data sets into separate Neo4j databases. We ran first and third LUBM queries (Q1 and Q3) on each of the systems. In the case of Neo4j we formulated the two LUBM queries as Cypher queries. In the case of Acacia-RDF we used the SPARQL syntax. We have plot the results in Figure 6. Note that all the performance values listed in this paper (except Neo4j non-cached scenarios) are three times averages taken after running multiple warm up runs of the same query. In the case of Neo4j non-cached scenario, when we obtain the performance numbers for the first query, we first ran third query ten times and then ran first query. This was to avoid Neo4j’s query caching feature which made third query to run without the effect of caching. This leads to fair comparison between the two systems when they do not employ any caching. The corresponding result is shown in the curve Neo4j-Q1. The results obtained by just running Neo4j three times without such technique is shown in Neo4j-Q1-Without-Caching. We did similar experiment for LUBM Q3 scenario as well.

<table>
<thead>
<tr>
<th>ID</th>
<th>Data set name</th>
<th>Vertices</th>
<th>Edges</th>
<th>File size</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
<td>LUBM-5</td>
<td>0.10M</td>
<td>0.83M</td>
<td>51.9MB</td>
</tr>
<tr>
<td>G2</td>
<td>LUBM-10</td>
<td>0.21M</td>
<td>1.70M</td>
<td>105.9MB</td>
</tr>
<tr>
<td>G3</td>
<td>LUBM-20</td>
<td>0.44M</td>
<td>2.59M</td>
<td>224.8MB</td>
</tr>
<tr>
<td>G4</td>
<td>LUBM-40</td>
<td>0.86M</td>
<td>7.10M</td>
<td>445MB</td>
</tr>
<tr>
<td>G5</td>
<td>LUBM-80</td>
<td>1.7M</td>
<td>14.33M</td>
<td>862MB</td>
</tr>
<tr>
<td>G6</td>
<td>LUBM-160</td>
<td>3.6M</td>
<td>28.50M</td>
<td>1.70G</td>
</tr>
</tbody>
</table>

In the second experiment we conducted a scalability experiment of LUBM queries with varying number of X10 places. The objective was to observe the scalability of LUBM query execution. The results are shown in Figures 7 (single variable queries) and 8 (multi-variable query).

Finally, we evaluated the fault tolerance mechanism of Acacia-RDF. For this we choose G5 data set which is one of the two larger data sets used in our experiments. We observed the system characteristics when the system run Q2 with 16 places and when the system run Q2 with one place crashed (only having 15 alive places). The elapsed time (three times average) of nonfaulty execution was 51.2 seconds while with
one place killed it took 53.4 seconds of execution. In both the cases we receive the correct result for executing LUBM Q2.

VI. DISCUSSION

In this paper we described the system architecture of Acacia-RDF which is a distributed RDF graph database engine. We have made a performance study of Acacia-RDF comparing its performance to Neo4j using LUBM RDF benchmark. We also conducted a scalability analysis (strong scaling) of the Acacia-RDF with SPARQL queries. Finally, we investigated the impact of failure of a place for execution of a multi-variable query (Q2).

From the first category of the experiments we observed that Acacia-RDF outperforms Neo4j in certain scenarios when both the systems operate without caching mechanisms’ help for graphs less than LUBM 20. However, Acacia-RDF’s execution time rises along with the number of universities in the input LUBM data. We conducted Nmon [19] based profiling of the experiments and observed that the communication between the master and worker increases with the size of the LUBM data set. We are currently working on to reduce this communication overhead so that we could reduce the elapsed time of LUBM query execution.

From the second category of experiments we observed that Acacia-RDF’s scalability in a distributed environment with first and third LUBM queries running with 16 places on LUBM 10 data set with elapsed times of approximately 2 seconds. Furthermore, Acacia-RDF reported less than ten seconds elapsed times on 16 places for running the first three queries of the LUBM benchmark on $G_6$.

Overall through these experiments we observed that Acacia-RDF is work with data sets with much larger data sets in future (beyond LUBM 160).
VII. CONCLUSION AND FURTHER WORK

Scalable Linked data management has become a significant problem in recent times. In this paper we describe the system architecture of Acacia-RDF which is a distributed RDF graph database engine aimed to solve the problem of RDF data storage and querying in heterogeneous clouds. We have developed Acacia-RDF following a vertex-centric paradigm. Acacia-RDF stores its data in secondary storage and it has a scalable SPARQL processor which executes SPARQL queries concurrently. Acacia-RDF is developed in Managed X10. Furthermore, in this paper we have discussed how fault tolerance has been implemented in Acacia leveraging X10’s resiliency framework. We observed scalability of Acacia-RDF by running in a cluster of four computers up to 16 X10 places with first and third LUBM queries running on LUBM 10 data set with elapsed times of approximately 2 seconds. Acacia-RDF reported less than ten seconds elapsed times on 16 places for running the first and the third queries of the LUBM benchmark on 160 universities data set of 1.7GB in size. Acacia-RDF is one of the first such implementations of a distributed, scalable SPARQL data store and a processor completely developed in a PGAS language.

One of the original system design goals of Acacia was the operation in hybrid cloud environments. In future we hope to extend Acacia-RDF also to the domain of hybrid clouds. We also work on improving the fault tolerance mechanism of Acacia-RDF concentrating on failure of Place 0 scenario. We are also implementing several other graph algorithms such as K-Core on top of Acacia system. We hope to extend Acacia-RDF into the domain of time evolving graphs in future. We are also working on to compare Acacia-RDF with other famous Java based distributed graph databases such as Titan.

REFERENCES


