Distributed RDF Triple Store Using HBase and Hive

Albert Haque
University of Texas at Austin
ahaque@cs.utexas.edu

Lynette Perkins
University of Texas at Austin
lperkins21@gmail.com

December 18, 2012

ABSTRACT

The growth of web data has presented new challenges regarding the ability to effectively query RDF data. Traditional relational database systems efficiently scale and query distributed data. With the development of Hadoop its implementation of the MapReduce Framework along with HBase, a NoSQL data store, the semantics of processing and querying data has changed.

Given the existing structure of SPARQL, we devised and are continuing to develop a means to query RDF data effectively. By using HBase as a data store and HiveQL as the query generator, we implement a prototype intermediate translator which will generate HiveQL given a SPARQL query. This HiveQL query will then return data from our distributed HBase store.

1 INTRODUCTION

As the amount of data grows, database systems must query and process this data in an efficient manner. The current relational model has worked for many years, but now the immense amounts of data are beginning to stress these previous models. New fields such as business analytics and bioinformatics attempt to query and store tera- and petabytes of data [4].

2 BACKGROUND

We present a brief overview of the technologies used, present syntax examples, and how the systems are related on the Apache software stack.

2.1 MapReduce & Hadoop

MapReduce allows large tasks to be run in parallel by *mapping* key-value pairs to an intermediate set of key-value pairs which are then *reduced* by merging all values with the same intermediate key. [2]. Hadoop runs on the Hadoop Distributed File System (HDFS) and is written in Java.

When a MapReduce job is run on Hadoop, it is given to a *JobTracker* node who then gives the job to available *TaskTracker* nodes. The system is fault tolerant and if a job fails, it is reassigned and recomputed on another node [10].

2.2 RDF & SPARQL

Resource Description Framework, more commonly known as RDF, is a standard model for data interchange on the internet [11, 12]. RDF explains how two things are related through a triple. A triple takes the form of a subject s, object o, and predicate p – commonly written as (s, p, o). SPARQL (SPARQL Protocol and RDF Query

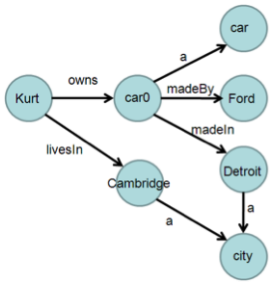

Figure 1: Small Graph of Triple Data [13]

Language), is a used to query RDF data. URIs are Uniform Resource Identifiers and are used to specify a dataset. For example, `PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>` will map any reference of bsbm: to the fu-berlin dataset. A SPARQL query can have multiple URIs in the same query. Similar to SQL, SPARQL queries follow a general format:

SELECT a FROM b WHERE c OPTIONAL d

The SELECT, FROM, and WHERE keywords are similar to SQL. Since it is not guaranteed that all data may contain the same attributes, we may want to retain results even if they do not satisfy a certain property. We call this property OPTIONAL.

Storing triples present a new challenge. How do we incorporate the dynamic nature of the data into our schema? New subjects, objects, and predicates are constantly being added. A RDBMS would not be appropriate since the final schema is not known. Researchers at various institutions have developed triple-stores to house RDF data. We will explore these methods later in this paper.

2.3 HBase

HBase is built on top of the Hadoop Framework and is based on Google Bigtable. It provides a scalable way to store data in a column oriented design. HBase is becoming ubquitous in the big data field as a means to capture real time data based on several factors.

An HBase value V_1 is located by a row key r, column family f, column name c, and optionally a timestamp t. We can identify V_1 with by (r_1, f_1, c_1, t_1). Inheriting from BigTable, HBase does not require any space for storing null values [1]. Figure 1 gives an overview of how HBase stores data, logically, physically, and on disk. Columns within column families are condensed before being written to disk, timestamps are sorted such that the most recent value appears first, and null values simply do not appear on disk.

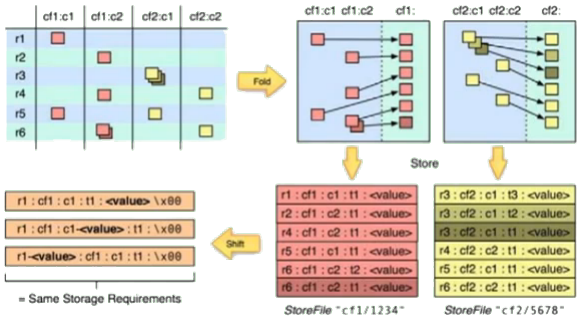

Figure 2: Logical to Physical Model (HBase) [5]

Column families allow us to take advantage of sngatial locality by keeping columns within a column family close together on disk. We can also treat column families as different entities by giving them different properties such as compressing column family A but not B.

Since values are identified by their unique set of identifiers (r, f, c, t), it is possible to have only one column containing values, identified by its row key – this is known as a tall-narrow table [8]. The opposite would be a flat-wide table where we try to minimize the number of rows. The two methods require identical disk space however the difference is how we interact with the data. For example, consider the case where we have a flat-wide table and row key r_1 contains values in columns $c_1, c2, \ldots, cn$. Deleting row r_1 would require one operation. However, in a tall-narrow table, the deletion requires n different operations.

2.4 Hive

Hive is a data warehouse system that allows querying of systems employing HDFS. It was invented by Facebook and is currently used on their Messages service and is offered by Amazon Web Services [6, 15]. Using a SQL-like language called HiveQL, it allows MapReduce jobs including joins [16]. Since Hive runs atop the Hadoop platform, queries may take minutes to hours to complete.

Hive is linked to HBase by mapping columns in Hive to columns in HBase. Hive supports accessing different column families within HBase.

3 DESIGN

3.1 Schema

Given HBase as a data storage container, mapping RDF Data to HBase requires a schema design unique to HBase to manage the dataset given the differences between relational databases and NoSQL databases. It is paramount within the design, to establish column families within the initial design process, due to high cost of adding or modifying column families once data has been loaded into the database.

Also paramount in modeling RDF data is designating the row key, in HBase the row key is similar to a primary key in SQL databases. Having a unique row key will ensure quick access to the given set of data.

At this point, the rowkey is the subject of the data set, the predicate is the column name and the object is the value associated with row key and column for this particular column family.

Given an RDF triple T_i, $T_i = (s_i, p_i, o_i)$ for data set D. An HBase table RDF would exist with Column Family D where the row key is s_i, the column name is p_i and the value is o_i.

This is an initial and basic schema design for a given data set D, there are limitations based on this design including the inefficiency related to scanning all rows for a given column family. Perhaps a more promising but yet to be explored, would be to design the row key value differently where the row key would not simply be the subject but would be the concatenation of two of the three values from RDF Triple Ti such that the row key $= (s_i, p_i), (s_i, o_i), (p_i, o_i), (p_i, s_i), (o_i, s_i) or (o_i, p_i)$. This is a hexastore approach to handling RDF triples [9].

In an attempt to accommodate multiple heterogeneous datasets, we use the column family to specify the dataset the user wishes to query. This fits nicely with the intended purpose of column families. Datasets will be stored relatively close to each other (on disk and node-wise). In addition, we can apply certain restrictions or properties to different column families depending on the requirements of the dataset.

There are several potential points of exploration, but the data model used is vitally important because it will affect the users approach to querying RDF data.

3.2 SPARQL Translation

Given the basic data model where subject represents the row key. Data can be processed and bulk loaded into HBase using an HBase API. One of the

leading methods for querying HBase tables is Hive, a query engine that sits on top of Hadoop and Hbase which runs HiveQL, a SQL-like language. HiveQL converts the SQL-like syntax into a series of map-reduce jobs.

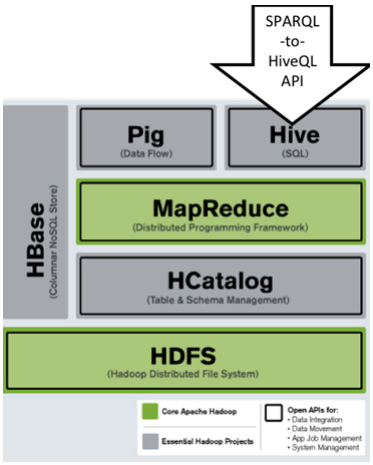

Figure 3: SPARQL to HiveQL Translator within the Hadoop Ecosystem

Using Hive, we began work on a SPARQL to HiveQL Translator API, which would work on top of Hive as show in Figure 2, which is a representation of the Hadoop Ecosystem.

We attempt to convert the BSBM Query 6 from SPARQL to HiveQL.

```
1  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-
       schema#>
2  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf
       -syntax-ns#>
3  PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de
       /bizer/bsbm/v01/vocabulary/>
4
5  SELECT ?product ?label
6  WHERE {
7      ?product rdfs:label ?label .
8      ?product rdf:type bsbm:Product .
9      FILTER regex(?label, "%word1%")
10 }
```

Listing 1: BSBM Query 6 (Step 1)

Given a SPARQL query Q_i, the translator converts Q_i into a logical model, then converts the logical model into an HBase expression tree and finally into HiveQL. A basic representation of this translator is shown in Figure 2, which displays the flow from a SPARQL query to HiveQL.

Given a SPARQL query, we devise a set of rules which can be applied to transform SPARQL to HiveQL. Some of the rules are listed in Table 1. Once we have applied these rules to the SPARQL query, we obtain an abstract SPARQL expression tree as shown in Figure 4. The SPARQL expression tree is then transformed to an HBase expression tree.

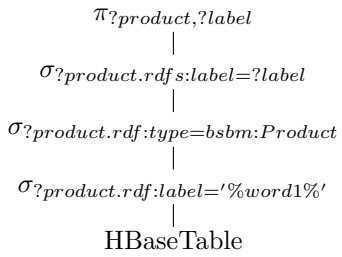

Figure 4: SPARQL Expression Tree (Step 2)

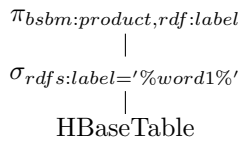

Figure 5: HBase Expression Tree (Step 3)

The HBase expression tree (Figure 5) is then transformed into HiveQL (Listing 2). It is possible to perform an intermediate step between the syntax to expression tree conversion and simplify the tree. But for simplicity, we leave all expression trees as they are for our SPARQL-to-HiveQL conversion. We can now run the BSBM Query 6 on HBase.

```
1 SELECT product, label
2 FROM HBaseTable
3 WHERE label LIKE "%word1%"
```

Listing 2: HiveQL Query (Step 4)

There is a significant amount of work related to translating SPARQL queries to SQL from Ultrawrap [17]. Given this rich source of content, with more extensive research into Ultrawrap, there is an opportunity to create a more robust SPARQL-to-HiveQL Translator API.

3.3 Related Efforts

A group at Raytheon BBN Technologies has developed a Scalable, High-Performance, Robust and Distributed (SHARD) triple store [13]. This runs on Hadoop and stores triples on HDFS in text files with one subject per text file line. This is similar to how HBase stores its data on disk. SHARD performs multiple MapReduce iterations over the different clauses. SHARD claims to be comparable to most industrial triple stores.

Sapphire leverages HBase to store triples using the basic graph pattern $(?s, ?p, ?o)$ [18]. It performs extremely well for queries with defined subjects. Although the schema only supports one BGP, it may be possible to incorporate a hexastore approach into our schema to allow querying of various BGPs.

Rule	SPARQL Syntax	Logical	HiveQL Syntax
1	SELECT ?a ?b ... ?z	$\pi_{?a,?b,...,?z}$	SELECT a, b, ..., z
2	SELECT ?a AS b	$\rho_{b/?a}$	SELECT ?a AS b
3	SELECT DISTINCT ?a	$\delta(\pi_{?a})$	SELECT DISTINCT ?a
4	WHERE {?s f:p o}	$\sigma_{?s.f:p=o}$	WHERE {?s.f:p = o}
5	WHERE {?s f:p ?o}	$\sigma_{?s.f:p=?o}$	WHERE {?s.f:p = ?o}
6	WHERE {FILTER regex(?a, string)}	$\sigma_{?a=string}$	WHERE ?a LIKE string
7	ORDER BY ?a	$\tau_{?a}$	ORDER BY ?a

Table 1: Rules for Transforming SPARQL to HiveQL

4 Future Work

This paper is an initial discussion and developmental process of storing RDF data using HBase, using HiveQL as a query generator and a SPARQL-to-HiveQL translator API to run queries on RDF data. This is the groundwork and basis for a more robust system for storing and querying RDF data. Areas of improvement include: (i) optimizing the SPARQL-to-HiveQL translator, (ii) refining the data model, and (iii) robust benchmark testing beginning with the Berlin Benchmark Testing.

As the Hadoop Ecosystem continues to evolve, an opportunity exists to continue to research and refine the approach used to store and query RDF data.

References

[1] Chang, F. et al., "Bigtable: A Distributed Storage System for Structured Data," in *OSDI*, 2006.

[2] Dean, J., Ghemawat, S., "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI*, 2006.

[3] Leo S, Santoni F, Zanetti G., "Biodoop: Bioinformatics on Hadoop," in *International Conference on Parallel Processing Workshops,* 2009.

[4] Taylor, R., "An Overview of the Hadoop/MapReduce/HBase Framework and its Current Applications in Bioinformatics," in *Bioinformatics Open Source Conference*, 2010.

[5] George, L., (2011, Oct 14) *HBase Schema Design - Things You Need to Know* [Video]. Available: `http://www.youtube.com/watch?&v=_HLoH_PgrLk`

[6] Thusoo, A., et al., "Hive: A Warehousing Solution Over Map-Reduce Framework," in *Very Large Data Bases*, 2009.

[7] Aiyer, A., et al., "Storage Infrastructure Behind Facebook Messages Using HBase at Scale," Facebook. 2012.

[8] Shreenivas, S. (2011, Sep 01) Tall-Narrow vs. Flat-Wide Tables [Online].

[9] Weiss, C., Karras, P., and Bernstein, A., "Hexastore: Sextuple Indexing for Semantic Web Data Management," in *Very Large Data Bases*, 2008.

[10] Apache Hadoop [Online]. Available: `http://wiki.apache.org/hadoop/`

[11] SPARQL Query Language for RDF, Available: `http://www.w3.org/TR/rdf-sparql-query/`

[12] Resource Description Framework (RDF), Available: `http://www.w3.org/RDF/`

[13] Rohloff, K., "Cloud computing for Scalability: The SHARD Triple-Store," [Webcast]. 2011.

[14] Rohloff, K., and Schantz, R., "High-Performance, Massively Scalable Distributed Systems using the MapReduce Software Framework: The SHARD Triple-Store," in *International Workshop on Programming Support Innovations for Emerging Distributed Applications*, 2010.

[15] Amazon, "Running Hive on Amazon Elastic MapReduce," 2012. `http://aws.amazon.com/articles/2857`

[16] Hive. Apache Software Foundation, 2012. `http://hive.apache.org/`.

[17] Sequeda, J., Depena, R., and Miranker, D., "Ultrawrap: Using SQL Views for RDB2RDF," a poster in the 8th International Semantic Web Conference, 2009.

[18] Savjani, N., "Sapphire: A distributed, scalable, column-oriented RDF Store on HBase," 2010.