5-2011

Distributed Semantic Web data management in HBase and MySQL cluster

Craig M. Franke
University of Texas-Pan American

DISTRIBUTED SEMANTIC WEB DATA MANAGEMENT

IN HBASE AND MYSQL CLUSTER

A Thesis

by

CRAIG M. FRANKE

Submitted to the Graduate School of the
University of Texas-Pan American
In partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2011

Major Subject: Computer Science

DISTRIBUTED SEMANTIC WEB DATA MANAGEMENT

IN HBASE AND MYSQL CLUSTER

A Thesis
by
CRAIG M. FRANKE

COMMITTEE MEMBERS

Dr. Artem Chebotko
Chair of Committee

Dr. John Abraham
Committee Member

Dr. Pearl Brazier
Committee Member

Dr. Richard Fowler
Committee Member

May 2011

ABSTRACT

Franke, Craig M., Distributed Semantic Web Data Management in HBase and MySQL Cluster. Master of Science (MS), May, 2011, 36 pp., 3 tables, 11 figures, references, 34 titles.

Various computing and data resources on the Web are being enhanced with machine-interpretable semantic descriptions to facilitate better search, discovery and integration. This interconnected metadata constitutes the Semantic Web. Efficient management of Semantic Web data, expressed using the W3C's Resource Description Framework (RDF), is crucial for supporting new data-intensive, semantics-enabled applications. In this work, we study and compare two approaches to distributed RDF data management based on emerging cloud computing technologies and traditional relational database clustering technologies. In particular, we design distributed RDF data storage and querying schemes for HBase and MySQL Cluster and conduct an empirical comparison of these approaches on a cluster of commodity machines using datasets and queries from the Third Provenance Challenge and Lehigh University Benchmark. Our study reveals interesting patterns in query evaluation, shows that our algorithms are promising, and suggests that cloud computing has a great potential for scalable Semantic Web data management.

ACKNOWLEDGMENTS

I thank my thesis advisor, Dr. Artem Chebotko, for his guidance over the course of my thesis work. My thanks also goes to Dr. John Abraham and Dr. Pearl Brazier both for their input and time as committee members and for their assistance in procuring the hardware for the distributed cluster used in my research. I also thank Dr. Richard Fowler for his input and time as a committee member.

TABLE OF CONTENTS

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER I

INTRODUCTION

The World Wide Web Consortium (W3C) has recommended and standardized a number of principles, languages, frameworks and best practices to interconnect various metadata into a next-generation web – the Semantic Web [1], [2]. The W3C's metadata acquisition languages include Resource Description Framework (RDF) [3], [4], RDF in attributes (RDFa) [5], RDF Schema (RDFS) [6], and Web Ontology Language (OWL) [7]. Government, academia, and industry actively embrace these technologies for capturing and sharing metadata on the Semantic Web. Just to name a few examples, oeGOV is making and publishing OWL ontologies for e-Government, U.S. census data is being published in RDF, bioinformaticians maintain the Universal Protein Resource (UniProt) in RDF, geoscientists publish worldwide geographical RDF database GeoNames, the largest electronics retailer in the U.S., BestBuy, publishes its full catalog in RDF, the largest social networking provider in the U.S., Facebook, embeds metadata in its webpages using RDFa, and the services computing community enhances existing Web services with semantic annotations using vocabularies, such as Semantic Markup for Web Services (OWL-S), Web Service Semantics (WSDL-S), and Semantic Web Services Ontology (SWSO).

The RDF data model is a directed, labelled graph that can also be serialized and viewed as a set of triples. A running example in this paper includes 10 triples that describe the authors using the Lehigh University Benchmark (LUBM) vocabulary [8] as shown in Figure 1. Each

triple consists of a subject, predicate, and object and defines a relationship between a subject and an object. In the figure, <> and "" denote resource identifiers and literals of some data type, respectively. For example, the first three triples state that a resource with identifier *C* is a *Student*, has name *Craig* and is a member of *IEEE*. This sample dataset can be queried using SPARQL [9] – a standard query language for RDF. SPARQL uses triple patterns and graph patterns that are matched over RDF data. For example, query Q14 from LUBM contains one triple pattern *?X* <*type*> <*UndergraduateStudent*> that returns all undergraduate student identifiers as bindings of variable *?X*. More details on SPARQL features and semantics can be found in [9], [10].

```
<C>        <type>          <Student>
<C>        <name>          "Craig"
<C>        <memberOf>      <IEEE>
<S>        <type>          <Student>
<S>        <name>          "Sam"
<S>        <memberOf>      <ACM>
<A>        <type>          <Faculty>
<A>        <name>          "Artem"
<A>        <memberOf>      <IEEE>
<A>        <memberOf>      <ACM>
```

Figure 1: Sample RDF triples.

With the rapid growth of the Semantic Web and widespread use of RDF as the primary language for metadata, efficient management of RDF data will become crucial for supporting new semantics-enabled applications in various domains. Many researchers have proposed using relational databases to store and query large RDF datasets. Such systems, called relational RDF databases or relational RDF stores [11], are now frequently in production. More recently, distributed technologies that are often used in cloud computing, such as Hadoop [12] and Hbase [13], are being explored for distributed and scalable RDF data management [14], [15].

To our best knowledge, this work provides the first performance comparison of the two worlds using our design and algorithmic solutions for storing and querying RDF data in HBase and MySQL Cluster.

The main contributions of our work are: (i) a novel database schema design for storing RDF data in HBase, (ii) efficient algorithms for SPARQL triple and basic graph pattern matching in HBase according to our schema, (iii) efficient SPARQL-to-SQL translation algorithm that results in flat SQL queries over our schema in MySQL Cluster, and (iv) empirical comparison of the proposed HBase and MySQL Cluster approaches for efficient and scalable storing and querying of Semantic Web data. Our work reveals interesting patterns in query evaluation, shows that our algorithms are promising, and suggests that cloud computing has a great potential for scalable Semantic Web data management.

The organization of this work is as follows. Related work is discussed in Chapter 2. Our design and algorithms for distributed RDF data storage and querying in HBase and MySQL Cluster are presented in Chapter 3 and 4, respectively. The performance study of the two approaches using datasets and queries from the Third Provenance Challenge and Lehigh University Benchmark is reported in Chapter 5. Finally, our concluding remarks are given in Chapter 6.

CHAPTER II

RELATED WORK

Besides Hbase [13], which is an open-source implementation of Google's Bigtable [16], there are multiple projects under the Apache umbrella that focus on distributed computing, including Hadoop, Cassandra, Hive, Pig, and CouchDB. Hadoop implements a MapReduce software framework and a distributed file system. Cassandra blends a fully distributed design with a column-oriented storage model and supports MapReduce as one of its features. Hive deals with data warehousing on top of Hadoop and provides its own Hive QL query language. Pig is geared towards analyzing large datasets through use of its high-level Pig Latin language for expressing data analysis programs, which are then turned into MapReduce jobs. CouchDB is a distributed, document-oriented, non-relational database that supports incremental MapReduce queries written in JavaScript. Along the same lines, other projects in academia and industry include Cheetah [17], Hadoop++ [18], G-Store [19], HadoopDB [20], and a distributed B-tree storage scheme [21].

Several related works on distributed RDF data management are briefly discussed in the following. Techniques for evaluating SPARQL basic graph patterns using MapReduce are presented in [14] and [22]. Efficient approaches to analytical query processing and distributed reasoning on RDF graphs in MapReduce-based systems are proposed in [23] and [24], respectively. RDF query processing in peer-to-peer environments is studied in [25] and [26], and mediation techniques for federated querying of distributed RDF sources are reported in [27] and

[28]. Use of HBase for text indexing is described in [29]. While the SPIDER system [30] that uses HBase for RDF query processing and the HBase extension for Jena [31] are announced, no details are reported. Finally, previous work [15] presents our initial findings on RDF data management in HBase. This work, when compared to [15] proposes a new, more effective HBase database schema design, more efficient algorithms for SPARQL triple and basic graph pattern matching, and an empirical comparison with a distributed relational RDF database. Our experimental comparison with [15] (not reported in the thesis) showed several orders of magnitude speedup for some queries and substantial improvements in scalability.

CHAPTER III

DISTRIBUTED RDF DATA STORAGE AND QUERYING IN HBASE

HBase stores data in tables that can be described as sparse multidimensional sorted maps and are structurally different from relations found in conventional relational databases. An HBase table (hereafter "table" for short) stores data rows that are sorted based on the row keys. Each row has a unique row key and an arbitrary number of columns, such that columns in two distinct rows do not have to be the same. A full column name (hereafter "column" for short) consists of a column family and a column qualifier (e.g., *family:qualifier*), where column families are usually specified at the time of table creation and their number does not change and column qualifiers are dynamically added or deleted as needed. A column of a given row, which we denote as table cell, can store a list of timestamp-value pairs, where timestamps are unique in the cell scope and values may contain duplicates. Rows in a table can be distributed over different machines in an HBase cluster and searched using two basic operations: (1) table scan and (2) retrieval of row data based on a given row key and, if available, columns and timestamps. Given that the table scan access path is inefficient for large datasets, the row key-based retrieval is the best feasible choice.

The sparse nature of tables makes them an attractive storage alternative for RDF data. RDF graphs are usually sparse as well: different resources are annotated with different properties and some annotations may not be stated explicitly due to inference. To support efficient retrieval of RDF data from tables in HBase, the basic querying constructs of SPARQL, such as triple

patterns, should be considered. At the very minimum, the database should support retrieval of RDF triples based on values of their subjects, predicates, objects, and their arbitrary combination.

T_{sp}

s	p:type	p:name	p:memberOf	...
<C>	{<Student>}	{"Craig"}	{<IEEE>}	...
<S>	{<Student>}	{"Sam"}	{<ACM>}	...
<A>	{<Faculty>}	{"Artem"}	{<IEEE>,<ACM>}	...

T_{op}

o	p:type	p:name	p:memberOf	...
<Student>	{<C>,<S>}			...
<Faculty>	{<A>}			...
"Craig"		{<C>}		...
"Sam"		{<S>}		...
"Artem"		{<A>}		...
<IEEE>			{<C>,<A>}	...
<ACM>			{<S>,<A>}	...

Figure 2: Storage schema and sample instance in HBase.

We propose to use a database schema with two tables to store RDF triples as shown in Figure 2. Table T_{sp} stores triple subjects as row keys, triple predicates as column names and triple objects as cell values. Table T_{op} stores triple objects as row keys, triple predicates as column names and triple subjects as cell values. Figure 2 shows a two-dimensional graphical representation of these tables with our sample RDF triples (see Figure 1) stored. In the figure, *s* and *o* denote row keys rather than columns; *type*, *name*, and *memberOf* are column qualifiers that belong to the same column family *p*; { } denote sets of cell values with timestamps omitted. More precisely, the structure of the rows can be shown using JavaScript Object Notation (JSON) as shown in Figure 3.

```
//the first row of Tsp
<C>: {
 p: {
   type:       { t1: <Student> },
   name:       { t2: "Craig" },
   memberOf:   { t3: <IEEE> }
 }
}

//the first row of Top
<Student>: {
 p: {
   type:       { t4: <C>,
                 t5: <S> }
 }
}
```

Figure 3: Structure of Rows in Javascript Object Notation.

In the first row of T_{sp}, <*C*> is a row key, *p* is a column family, *type*, *name*, and *memberOf* are column qualifiers, t_1, t_2, and t_3 are timestamps, and the rest are values. The structure of the first row of T_{op} can be interpreted in a similar way but it should be noted that, while the graphical representation in Figure 2 shows blank values for some table cells, the row contains no information about such values or the respective columns. This illustrates the sparse storage nature of HBase tables and shows that no space is wasted.

The proposed schema requires that RDF data is stored twice - replication that contributes to the robustness of the system. Tables T_{sp} and T_{op} can be used to efficiently retrieve triples with known subjects and objects, respectively. Retrieval of triples based on a predicate value requires a scan of one of the tables, which may not be efficient. To try to remedy this problem, we could have created a table, i.e., T_{ps} or T_{po}, with predicates as row keys and subjects or objects as columns. However, such a solution can only provide marginal improvements, since the number of predicates in an ontology is usually fixed and relatively small, which implies that this new table can contain only a small number of large rows (one per distinct predicate) and retrieval of

any individual row is still expensive.

For HBase to be able to evaluate SPARQL queries, we design three functions that deal with triple patterns and basic graph patterns.

Our first function, matchTP-T, allows matching of a triple pattern over a triple and is outlined in Figure 4. It is a general-purpose function that depends on neither our storage schema nor HBase; it also appeared in [15]. matchTP-T takes a triple pattern *tp* and a triple *t* and returns *true* if they match or *false* otherwise. To check that *tp* matches *t*, several conditions must be satisfied: (1) a variable can match anything, (2) a URI or literal must match itself, and (3) a variable that occurs more than once must match the same term for all occurrences.

Algorithm matchTP-T: Matching a triple pattern over a triple

```
1: function matchTP-T
2: input: triple pattern tp = (sp, pp, op), triple t = (s, p, o)
3: output: true or false
4: if (tp.sp is a variable ∨ tp.sp = t.s) ∧ (tp.pp is a variable ∨ tp.pp =
   t.p) ∧ (tp.op is a variable ∨ tp.op = t.o) then
5:     if tp.sp = tp.pp ∧ t.s ≠ t.p then
6:         return false
7:     end if
8:     if tp.sp = tp.op ∧ t.s ≠ t.o then
9:         return false
10:    end if
11:    if tp.pp = tp.op ∧t.p ≠ t.o then
12:        return false
13:    end if
14:    return true
15: end if

16: return false
17: end function
```

Figure 4: Algorithm matchTP-T.

Function matchTP-DB as outlined in Figure 5 is used to match a triple pattern *tp* in an HBase database *DB* according to our storage schema with two tables. The output of this function is a bag (multi-set) *B* that holds all matching triples in the database.

Algorithm matchTP-DB: Matching a triple pattern over a database

```
1: function matchTP-DB
2: input: triple pattern tp = (sp, pp, op), database DB = {T_sp, T_op}
3: output: bag of triples B_(sp,pp,op) = {t|t is in DB ∧ t matches tp}
4: B = ø
5: if tp.sp is not a variable then
6:     if tp.pp is not a variable then
7:         Retrieve triples into bag B from T_sp where row key s = tp.sp
           using column tp.pp
8:     else
9:         Retrieve triples into bag B from T_sp where row key s = tp.sp
           using all columns
10:    end if
11:    Remove any triple t ∈ B from B if matchTP-T(tp, t) = false
12:    return B
13: end if

14: if tp.op is not a variable then
15:    if tp.pp is not a variable then
16:        Retrieve triples into bag B from T_op where row key o = tp.op
           using column tp.pp
17:    else
18:        Retrieve triples into bag B from T_op where row key o = tp.op
           using all columns
19:    end if
20:    Remove any triple t ∈ B from B if matchTP-T(tp, t) = false
21:    return B
22: end if

23: if tp.pp is not a variable then
24:    Retrieve triples into bag B from T_sp (or T_op) using column tp.pp
25: else
26:    Retrieve triples into bag B from T_sp (or T_op) using all columns
27: end if
28: Remove any triple t ∈ B from B if matchTP-T(tp, t) = false
29: return B
30: end function
```

Figure 5: Algorithm matchTP-DB.

The algorithm deals with three disjoint cases. First, if *tp*'s subject pattern is not a variable, the function retrieves matching triples from table T_{sp}, such that a row with key *tp.sp* is accessed. If *tp.pp* is not a variable, only values in the column with qualifier *tp.pp* are retrieved for this row;

otherwise, all columns must be retrieved. Triples are reconstructed from row keys, column qualifiers, and cell values and are placed into *B*. Since *tp.op* may not be a variable or it may be a variable that occurs twice in the triple pattern, matchTP-T is applied on all the triples to eliminate non-matching ones. After this filtering, triples in *B* are returned. Second, if *tp*'s object pattern is not a variable, the function retrieves matching triples from table T_{op} using a similar strategy. Finally, when both *tp.sp* and *tp.op* are variables, one of the tables must be scanned to retrieve all rows. If *tp.pp* is not a variable, non-matching columns are discarded; otherwise, values in all columns are used.

Our last function matchBGP-DB is outlined in Figure 7. It matches a SPARQL basic graph pattern *bgp* that consists of a set of triple patterns tp_1, tp_2, ..., tp_n over an HBase database and returns a relation with a bag *B* of graphs constituted by matching triples. The algorithm starts by ordering triple patterns in *bgp* using two criteria: (1) triple patterns that yield a smaller result should be evaluated first to decrease a number of iterations and (2) triple patterns that have a shared variable with preceding triple patterns should be given a preference over triple patterns with no shared variables to avoid unnecessary Cartesian products. As an example, consider Figure 6, which is a query from LUBM [8] and its reordered version.

// original query Q7 from LUBM
tp_1: `? X <type> <Student> .`
tp_2: `? Y <type> <Course> .`
tp_3: `<http://...Professor0> <teacherOf> ? Y .`
tp_4: `? X <takesCourse> ? Y .`

// reordered basic graph pattern
tp_3: `<http://...Professor0> <teacherOf> ? Y .`
tp_2: `? Y <type> <Course> .`
tp_4: `? X <takesCourse> ? Y .`
tp_1: `? X <type> <Student> .`

Figure 6: Original and reordered query 7 from LUBM

The order in the original query does not satisfy the desired criteria: tp_1 yields a large result set with all students across all universities in a dataset; tp_2 has no shared variables with tp_1 and a memory-expensive Cartesian product must be computed between tp_1's and tp_2's results. The reordered query can save both memory and network transfer time: not only is tp_3, the triple pattern with the smallest result, placed at the first position, but the Cartesian product is also eliminated.

Next, the algorithm evaluates the first triple pattern in ordered *bgp* using matchTP-DB. If the result in *B* is empty, the algorithm returns an empty result without evaluating subsequent triple patterns. Otherwise, matchBGP-DB iterates over other triple patterns computing either joins on shared variables or Cartesian products if no shared variables exist. Each join resembles the index-nested-loops join strategy known in relational databases. Instead of directly evaluating triple pattern tp_i using matchTP-DB, shared variables are first substituted with their bindings found in *B* and the resulting triple patterns *tp′* in set *TP* are evaluated using matchTP-DB. If *tp′* yields a non-empty result, triples in *B′* are concatenated with the corresponding triples in *B*; otherwise, previous solutions from *B* whose variable bindings were used in variable substitution to obtain *tp*' are removed as the join condition has failed.

Algorithm matchBGP-DB: Matching a basic graph pattern over a database

```
1: function matchBGP-DB
2: input: basic graph pattern bgp = {tp1, tp2, …, tpn-1, tpn} and
   n ≥ 1, database DB = {Tsp, Top}
3: output: bag of tuples B(tp1.sp,tp1.pp,tp1.op,tp2.sp,...) = {g|g is a graph in DB
   ∧ g matches bgp}
4: B = ø
5: Order triple patterns in bgp, such that triple patterns that yield a
   smaller result and triple patterns that have a shared vairable with
   preceding triple patterns should be evaluated first.
6: Let ordered bgp = (tp1, tp2, …, tpn)
7: B = matchTP-DB(tp1, DB)
8: if B = ø then return B end if
9: for each tpi in (tp2, … tpn) do
10:     if tpi has shared variables with tpi-1, … tp1 then
11:         Let TP be a set of triple patterns obtained by substituting
            shared variables with their respective bindings from B
12:         for each tp′ in TP do
13:             B′ = matchTP-DB(tp′, DB)
14:             if B′ ≠ ø then
15:                 Add triples in B′ to B by concatenating each triple t′
                    ∈ B′ with every tuple t ∈ B if t's bindings were used
                    in variable substitution to obtain tp′
16:             else
17:                 Remove any tuple t from B if t's bindings were used
                    in variable substitution to obtain tp′
18:                 if B = ø then return B end if
19:             end if
20:         end for
21:     else
22:         B′ = matchTP-DB(tpi, DB)
23:         Compute Cartesian product of B and B′, i.e B = B × B′
24:     end if
25: end for
26: return B
27: end function
```

Figure 7: Algorithm matchBGP-DB.

Other SPARQL constructs, such as projection (*SELECT*), filtering (*FILTER*), alternative graph patterns (*UNION*), and optional graph patterns (*OPTIONAL*) can be incorporated in the presented algorithmic framework, but is out of this work's scope.

CHAPTER IV

DISTRIBUTED RDF DATA STORAGE AND QUERYING IN MYSQL CLUSTER

Relational RDF databases use several approaches to database schema generation that include schema-oblivious, schema-aware, data-driven, and hybrid strategies [10]. These approaches feature various database relations, such as property, class, class-subject, class-object, and clustered property tables. In this work, we use a schema-oblivious approach that employs a generic schema with a single table $T(s,p,o)$, where columns s, p, and o store triple subjects, predicates, and objects, respectively. Figure 8 shows table T with our sample RDF triples (see Figure 1) stored.

Our rationale for choosing this schema is threefold. First, it can support ontology evolution with no schema modifications. The schema proposed for HBase is also very flexible as only column qualifiers may dynamically change and such changes are performed on the row level. Second, most mentioned tables employed by relational RDF databases can be viewed as a result of horizontal partitioning of table T. However, partitioning is already performed by MySQL Cluster automatically. Finally, this schema allows lossless storage and is easy to implement. In particular, it greatly simplifies SPARQL-to-SQL translation that is required to query stored RDF data.

T

s	**p**	**o**
<C>	<type>	<Student>
<C>	<name>	"Craig"
<C>	<memberOf>	<IEEE>
<S>	<type>	<Student>
<S>	<name>	"Sam"
<S>	<memberOf>	<ACM>
<A>	<type>	<Faculty>
<A>	<name>	"Artem"
<A>	<memberOf>	<IEEE>
<A>	<memberOf>	<ACM>

Figure 8: Storage schema and sample instance in MySQL Cluster.

To execute SPARQL queries over our database schema in MySQL Cluster, we present a SPARQL-to-SQL query translation algorithm for basic graph patterns. The algorithm is based on previous work [10] on semantics-preserving SPARQL-to-SQL translation, but it is optimized to generate flat SQL queries. Query flattening (vs. nesting) removes a concern of triple pattern reordering in basic graph patterns since a relational query optimizer is capable of selecting a "good" join execution order automatically.

The BGPtoFlatSQL function is outlined in Figure 9. It translates a SPARQL basic graph pattern *bgp* that consists of a set of triple patterns tp_1, tp_2, ..., tp_n into an equivalent flat SQL query that can be executed over a MySQL Cluster database with our schema. BGPtoFlatSQL constructs *from*, *where*, and *select* clauses of an SQL query as follows. For each triple pattern in *bgp*, a unique table alias is assigned and table *T* with this alias is appended to the *from* clause. The algorithm then computes an inverted index on all variables in *bgp*, such that each distinct variable is associated with attributes in the respective tables from the *from* clause. The *where* clause is first constructed to ensure that any non-variables in *bgp* are restricted to their values (e.g., literals or identifiers).

Algorithm BGPtoFlatSQL: Translation of SPARQL basic graph patterns to flat SQL queries

```
1: function BGPtoFlatSQL
2: input: basic graph pattern bgp = {tp1, tp2, …, tpn-1, tpn} and
   n ≥ 1, database DB = {T }
3: output: flat SQL query
4: Assign a unique alias ai to each triple pattern tpi ∈ bgp
5: select = “”; from = “”; where = “”
6: // Construct the SQL From clause:
7: for each tpi ∈ bgp do
8:     from += “T $ai, ”
9: end for
10: // Construct an inverted index (hash) h on variables in bgp:
11: for each tpi ∈ bgp do
12:     for each variable ?v found in tpi do
13:         Let p be “s”, “p”, or “o” if ?v is at the subject, predicate, or
            object position, respectively, in tpi
14:         h(?v) = h(?v) ∪ {“$ai..$p”}
15:     end for
16: end for
17: // Construct the SQL Where clause:
18: for each tpi ∈ bgp do
19:     for each instace or literal l found in tpi do
20:         Let p be “s”, “p”, or “o” if l is at the subject, predicate, or
            object position, respectively, in tpi
21:         where += “$ai..$p= '$l' And ”
22:     end for
23: end for
24: for each distinct variable ?v found in bgp and |h(?v)| > 1 do
25:     Let x ∈ h(?v)
26:     for each   y ∈ h(?v) and y ≠ x do
27:         where += “$x = $y And”
28:     end for
29: end for
30: // Construct the SQL Select clause:
31: for each distinct variable ?v found in bgp do
32:     Let x ∈ h(?v)
33:     Let m is the name of variable ?v
34:     Select += “$x As $m, ”
35: end for
36: return “Select $select From $from Where $where”
37: end function
```

Figure 9: Algorithm BGPtoFlatSQL.

The inverted index is then used to append join conditions into the *where* clause, such that all attributes that correspond to the same variable must be equal. Finally, the *select* clause is generated to include attributes that correspond to every distinct variable in *bgp*, with attributes being renamed as variable names. Figure 10 illustrates the result of a translation performed with BGPtoFlatSQL:

```
//input SPARQL query Q7 from LUBM
tp1: ?X <type> <Student> .
tp2: ?Y <type> <Course> .
tp3: <http://...Professor0> <teacherOf> ?Y .
tp4: ?X <takesCourse> ?Y .

//output equivalent SQL query
Select tp1.s As X, tp2.s As Y
From T tp1, T tp2, T tp3, T tp4
Where tp1.p = '<type>' And
      tp1.o = '<Student>' And
      tp2.p = '<type>' And
      tp2.o = '<Course>' And
      tp3.s = '<http://...Professor0>' And
      tp3.p = '<teacherOf>' And
      tp4.p = '<takesCourse>' And
      tp1.s = tp4.s And tp2.s = tp3.o And
      tp2.s = tp4.o
```

Figure 10: Translation of SPARQL to SQL with BGPtoFlatSQL

Translation of other SPARQL constructs into SQL is out of this work's scope; details can be found in [10].

CHAPTER V

PERFORMANCE STUDY

This chapter reports our empirical comparison of the proposed approaches to distributed Semantic Web data storage and querying in HBase and MySQL Cluster.

Experimental Setup

Hardware

Our experiments used nine commodity machines with identical hardware. Each machine had a late-model 3.0 GHz 64-bit Pentium 4 processor, 2 GB DDR2-533 RAM, 80 GB 7200 rpm Serial ATA hard drive and two (on-board and add-on) Ethernet adapters. The machines were networked together via their add-on gigabit Ethernet adapters connected to a Dell PowerConnect 2724 gigabit Ethernet switch and configured with static, non-routable IP addresses. The machine functioning as the master in the cluster utilized its on-board Ethernet adapter to function as a network gateway for accessing the cluster via SSH. The hard drives on each machine had one partition of 64 GB dedicated to running the experiments. The machines were all running 64-bit Debian Linux 5.0.7 and Oracle JDK 6. Experiment code was executed on the master machine for both HBase and MySQL Cluster.

Hbase

Hadoop 0.20.2, with a modified core library, and HBase 0.90 were used. The installation on each machine was configured to use the fully-distributed mode. The master machine ran the

Hadoop namenode and HBase master server. The other eight machines ran the Hadoop datanode and HBase region server/zookeeper. The Hadoop and HBase configuration of each machine began with the minimal amount of configuration information to construct the cluster while accepting all other configuration values as default. Minor changes to the default configuration beyond this for stability included setting each block of data to replicate two times and increasing the HBase max heap size to 1.2 GB.

MySQL Cluster

MySQL Cluster 7.1.9a was used. The cluster used a modified configuration based on the MySQL Cluster Quick Start Guide with increased memory available for use by NDB data nodes. The master machine ran the NDB management server and MySQL server components while the other eight machines ran as NDB data nodes. The machines were split into four node groups consisting of two replicas per node group, with each machine representing a replica. Experiment code utilized JDBC driver 5.1.14 for MySQL.

Our implementation

Our algorithms were implemented in Java and the experiments were conducted using Bash shell scripts to execute the Java class files and store the results in an automated and repeatable manner. These shell scripts were responsible for loading datasets of varying sizes and executing queries multiple times. Both HBase and MySQL Cluster experiments utilized near-identical shell scripts.

Datasets and Queries

The experiments used datasets from the Third Provenance Challenge (PC3) [32] and Lehigh University Benchmark (LUBM) [8]. PC3 employed the Load Workflow that was a variation of a workflow used in the Pan-STARRS project. Via simulation, a number of scientific

workflow provenance documents for multiple workflow runs was generated and represented using Tupelo's OWL vocabulary available from the Open Provenance Model website [33]. Each workflow execution generated approximately 700 RDF triples. Table 1 indicates the characteristics of each PC3 dataset.

Dataset	# of workflow runs	# of RDF triples	Disk space
*D*1	1	700	86 KB
*D*2	10	7,000	860 KB
*D*3	100	70,000	8.7 MB
*D*4	1,000	700,000	88 MB
*D*5	10,000	7,000,000	895 MB
*D*6	100,000	70,000,000	9 GB

Table 1: PC3 Dataset Characteristics.

The three PC3 SPARQL queries utilized for the experiments can be found in previous work [15]. LUBM is a popular benchmark for RDF databases that includes the OWL university ontology, RDF data generator, and 14 test queries. Table 2 indicates the characteristics of each generated LUBM dataset.

Dataset	# of universities	# of RDF triples	Disk space
*L*1	1	38,600	4.4 MB
*L*2	5	563,000	68 MB
*L*3	10	1,211,000	146 MB
*L*4	30	3,908,000	477 MB
*L*5	50	6,593,000	807 MB
*L*6	70	9,308,000	1.1 GB
*L*7	90	11,964,000	1.5 GB
*L*8	110	14,649,000	1.8 GB
*L*9	200	26,635,000	3.3 GB
*L*10	400	53,301,000	6.6 GB
*L*11	600	80,043,000	9.9 GB

Table 2: LUBM Dataset Characteristics.

The LUBM queries expressed in a KIF-like language can be found at the LUBM website [34]; for the purpose of our experiments, they were rewritten in SPARQL. Since our experiments tested query performance and not reasoning ability, each generated LUBM dataset was augmented with additional triples needed to produce the sample query results supplied by LUBM.

Data Ingest Performance

Multiple data loading methods were evaluated under both HBase and MySQL Cluster, including statement-by-statement, batch, and bulk load methods. Figure 11 only reports the best performers for each system. In particular, batch data loading in the chunks of 1,000 triples at a time was used in HBase and bulk loading was used in MySQL Cluster. Bulk loading is a new feature to HBase 0.90; it involved somewhat complicated formatting of the data, placement of the data in HDFS, and use of the map-reduce framework to convert the plain text data into the HBase binary data format. As a result, our simpler batch loading alternative implemented in Java showed better performance. Bulk loading in MySQL Cluster relied on the LOAD DATA INFILE statement that loaded large data files converted from the N-Triple format to the format preferred by MySQL.

The load times of PC3 and LUBM datasets are reported in Figure 11 with detailed data in the appendix. For the given data and index memory configuration, MySQL Cluster was able to load datasets up to *D*5 and *L*8. HBase successfully loaded all the datasets. MySQL Cluster initially demonstrated a significant advantage over HBase, however this performance advantage decreased with dataset size growth. For example, MySQL Cluster was 3 times faster than HBase on *L*1 and only 1.5 times faster on *L*8. It is possible that, with larger datasets, the advantage would be further reduced if not eliminated.

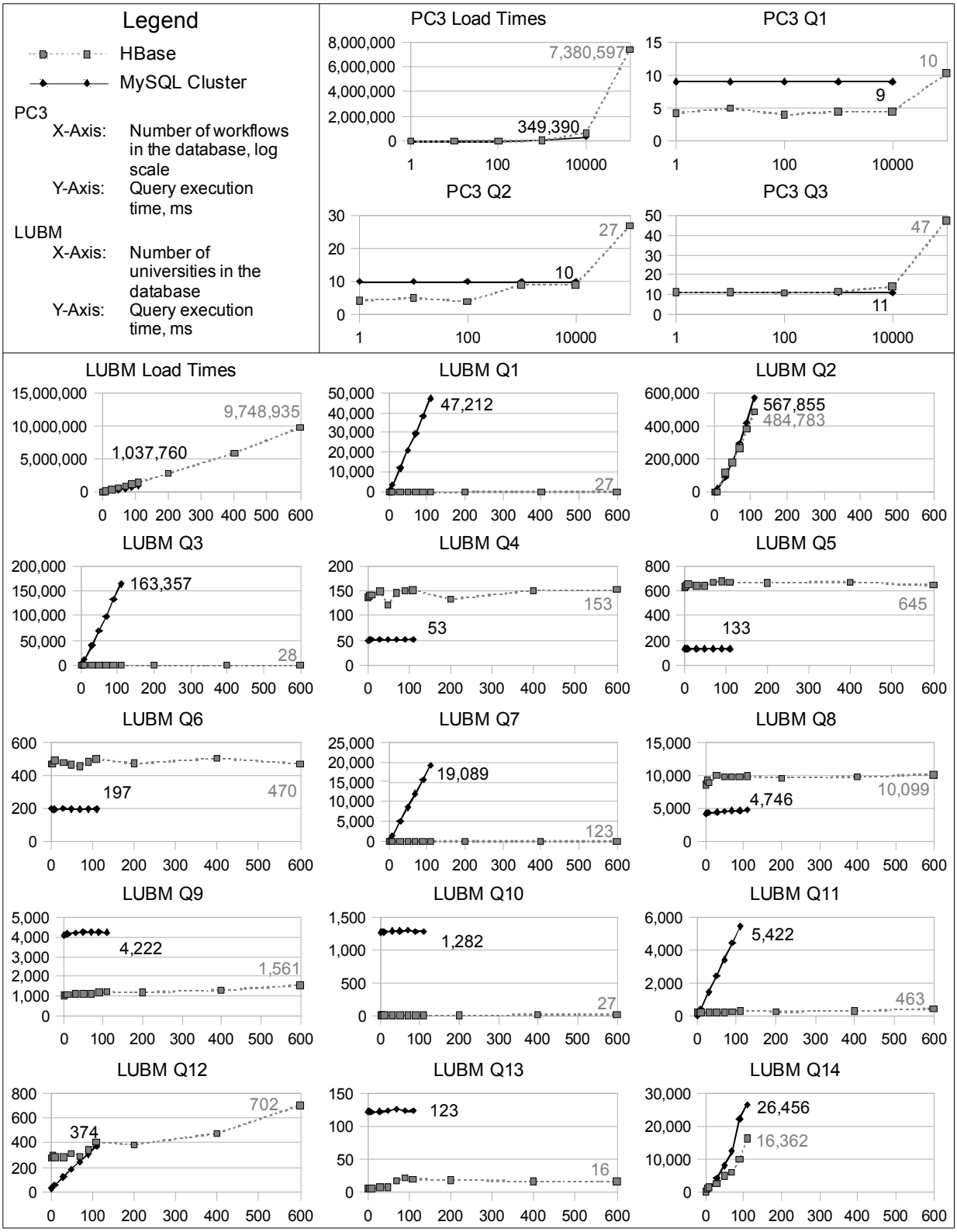

Figure 11: Data ingest and query performance and scalability.

It also should be noted that HBase stored twice as many triples with tables and as MySQL Cluster with only one table *T*. Overall, the data ingest performance showed to be efficient and revealed linear scalability for each system.

Query Evaluation Performance

HBase and MySQL Cluster query performance and scalability on PC3 and LUBM datasets are reported in Figure 11 with detailed data in the appendix. The PC3 benchmark used three queries with varying complexity: *Q*1 was the simplest query with one triple pattern, *Q*2 had three triple patterns, and *Q*3 was the most complex one consisting of six triple patterns. The basic graph patterns in all three queries returned a small result. Both HBase and MySQL Cluster showed very efficient and comparable response times, with the former being slightly faster. At *D*6, HBase took a slight upward turn in times that had previously remained nearly flat, which signifies that the graphs have a small slope (while the dataset size increased by a factor of 10, the response times increased by a factor of only around 2 to 4); similar behavior was also observed for some LUBM queries.

The LUBM benchmark used 14 queries whose complexities are shown in Table 3. LUBM query evaluation results for HBase and MySQL Cluster revealed several interesting patterns, denoted as *A*, *B*, *C*, *D*, and *E* in Table 3.

Query complexity (# of triple patterns)	**LUBM queries and their evaluation patterns**
1	*Q*6(*C*), *Q*14(*B*)
2	*Q*1(*A*), *Q*3(*A*), *Q*5(*C*), *Q*10(*D*), *Q*11(*A*), *Q*13(*D*)
3	N/A
4	*Q*7(*A*), *Q*12(*E*)
5	*Q*4(*C*), *Q*8(*C*)
6	*Q*2(*B*), *Q*9(*D*)

Table 3: LUBM Query Complexity and Evaluation Patterns.

Pattern *A* (*Q*1, *Q*3, *Q*7, and *Q*11) is characterized by the rapidly increasing query execution time for MySQL Cluster and nearly constant response time for HBase as the dataset size increased. Pattern *B* (*Q*2 and *Q*14) is characterized by rapid performance degradation in both systems. While *Q*2 had six triple patterns, *Q*14 had only one triple pattern that retrieved all undergraduate students across all universities in the database. Both queries yielded large results, such that results for *L*9, *L*10, and *L*11 could not fit into main memory on the HBase master server. In the case of *Q*14, which involved no joins, it is evident that the major factor in query performance is data transfer time and it is hardly possible to achieve better performance on the given hardware. Patterns *C* (*Q*4, *Q*5, *Q*6, and *Q*8) and *D* (*Q*9, *Q*10, and *Q*13) include queries whose performance showed limited or no growth in execution times with an increase in the data size in both systems. Pattern *C* queries were approximately 2 to 3 times faster on MySQL Cluster and pattern *D* queries were anywhere from 3 to 47 times faster on Hbase. Pattern *E* stands out on its own with a single representative query – *Q*12. For smaller datasets, *Q*12 was much faster on MySQL Cluster, however its performance quickly decreased on larger datasets, much like in pattern *A*. HBase, on the other hand, demonstrated a gradual increase in execution time: close to the 100 university mark, HBase performance exceeded MySQL Cluster performance.

The comparison of the query evaluation patterns and query complexity in LUBM (see Table 3) does not reveal any strong correlation between the two characteristics. The query complexity is not the sole indicator of query performance under HBase and MySQL Cluster: the size of intermediate and final results can have a significant impact.

Overall, in our experiments, the HBase approach showed better performance and scalability than the MySQL Cluster approach. Neglecting $Q2$ and $Q14$ of LUBM, which are expensive due to returning large results, the evaluation of two queries over the largest LUBM dataset in HBase took over 1s: $Q8$ (10s) and $Q9$ (1.5s). In contrast, six LUBM queries took over 1s in MySQL Cluster under similar circumstances. Finally, $Q1$, $Q3$, $Q7$, and $Q11$ of LUBM scaled significantly worse in MySQL Cluster.

Summary

Our performance study revealed interesting patterns in query evaluation, showed that our algorithms are efficient, and suggested that cloud computing has a great potential for scalable Semantic Web data management. Given that the experiments were performed with large datasets on commodity machines, both HBase and MySQL Cluster approaches showed to be quite efficient and promising. The proposed approaches were up to the task of efficiently storing and querying large RDF datasets. Overall, the experimental results were in favor of the HBase approach: not only were larger datasets able to load, but query performance and scalability were shown to be superior in many cases.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

In this work, we studied the problem of distributed Semantic Web data management using state of the art cloud and relational database technologies represented by HBase and MySQL Cluster. We designed a novel database schema for HBase to efficiently store RDF data and proposed scalable querying algorithms to evaluate SPARQL queries in HBase. We chose a generic RDF database schema for MySQL Cluster and presented a SPARQL-to-SQL translation algorithm that generates flat SQL queries for SPARQL basic graph patterns. Finally, we conducted an experimental comparison of the two proposed approaches on a cluster of commodity machines using datasets and queries of the Third Provenance Challenge and Lehigh University Benchmark. Our study concluded that, while both approaches were up to the task of efficiently storing and querying large RDF datasets, the HBase solution was capable of dealing with larger RDF datasets and showed superior query performance and scalability. We believe that cloud computing has a great potential for scalable Semantic Web data management.

In the future, we will focus on architectural aspects of an RDF database management system in the cloud, search for optimizations in schema design, explore additional SPARQL features, and research inference support in distributed environments.

REFERENCES

[1] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," *Scientific American*, May 2001.

[2] N. Shadbolt, T. Berners-Lee, and W. Hall, "The Semantic Web revisited," *IEEE Intelligent Systems*, vol. 21, no. 3, pp. 96–101, 2006.

[3] W3C, "RDF Primer. W3C Recommendation, 10 February 2004. F. Manola and E. Miller (Eds.)," 2004, available from http://www.w3.org/TR/rdf-primer/.

[4] ------, "Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, 10 February 2004. G. Klyne, J. J. Carroll, and B. McBride (Eds.)," 2004, available from http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[5] ------, "RDFa in XHTML: Syntax and Processing. W3C Recommendation, 14 October 2008. B. Adida, M. Birbeck, S. McCarron, and S. Pemberton (Eds.)," 2008, available from http://www.w3.org/TR/2008/REC-rdfa-syntax-20081014/.

[6] ------, "RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation, 10 February 2004. D. Brickley and R.V. Guha (Eds.)," 2004, available from http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[7] ------, "OWL Web Ontology Language Reference. W3C Recommendation, 10 February 2004. M. Dean and G. Schreiber (Eds.)," available from http://www.w3.org/TR/2004/REC-owl-ref-20040210/.

[8] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems." *Journal of Web Semantics*, vol. 3, no. 2-3, pp. 158–182, 2005.

[9] W3C, "SPARQL Query Language for RDF. W3C Recommendation, 15 January 2008. E. Prud'hommeaux and A. Seaborne (Eds.)." 2008, available from http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[10] A. Chebotko, S. Lu, and F. Fotouhi, "Semantics preserving SPARQL-to-SQL translation," *Data & Knowledge Engineering*, vol. 68, no. 10, pp. 973–1000, 2009.

[11] A. Chebotko and S. Lu, *Querying the Semantic Web: An Efficient Approach Using Relational Databases*. LAP Lambert Academic Publishing, 2009.

[12] *Apache Hadoop*, http://hadoop.apache.org.

[13] *Apache HBase*, http://hbase.apache.org.

[14] M. F. Husain, L. Khan, M. Kantarcioglu, and B. M. Thuraisingham, "Data intensive query processing for large RDF graphs using cloud computing tools," in *Proc. of the International Conference on Cloud Computing (CLOUD)*, 2010, pp. 1 – 10.

[15] J. Abraham, P. Brazier, A. Chebotko, J. Navarro, and A. Piazza, "Distributed storage and querying techniques for a Semantic Web of scientific workflow provenance," in *Proc. of the International Conference on Services Computing (SCC)*, 2010, 178-185.

[16] 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," *ACM Transactions on Computer Systems*, vol. 26, no. 2, 2008.

[17] S. Chen, "Cheetah: A high performance, custom data warehouse on top of MapReduce," *Proc. of the VLDB Endowment (PVLDB)*, vol. 3, no. 2, pp. 1459–1468, 2010.

[18] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, "Hadoop+ +: Making a yellow elephant run like a cheetah (without it even noticing)," *Proc. of the VLDB Endowment (PVLDB)*, vol. 3, no. 1, pp. 518–529, 2010.

[19] S. Das, D. Agrawal, and A. E. Abbadi, "G-Store: a scalable data store for transactional multi key access in the cloud," in *Proc. of the ACM Symposium on Cloud Computing (SoCC)*, 2010, pp. 163–174.

[20] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz, "HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads," *Proc. of the VLDB Endowment (PVLDB)*, vol. 2, no. 1, pp. 922–933, 2009.

[21] M. K. Aguilera, W. M. Golab, and M. A. Shah, "A practical scalable distributed B-tree," *Proc. of the VLDB Endowment (PVLDB)*, vol. 1, no. 1, pp. 598–609, 2008.

[22] J. Myung, J. Yeon, and S. Lee, "SPARQL basic graph pattern processing with iterative MapReduce," in *Proc. of the International Workshop on Massive Data Analytics on the Cloud (MDAC)*, 2010, pp. 6:1–6:6.

[23] P. Ravindra, V. V. Deshpande, and K. Anyanwu, "Towards scalable RDF graph analytics on MapReduce," in *Proc. of the International Workshop on Massive Data Analytics on the Cloud (MDAC)*, 2010, pp. 5:1–5:6.

[24] J. Urbani, S. Kotoulas, E. Oren, and F. van Harmelen, “Scalable distributed reasoning using MapReduce,” in *Proc. of the International Semantic Web Conference (ISWC)*, 2009, pp. 634–649.

[25] A. Matono, S. M. Pahlevi, and I. Kojima, “RDFCube: A P2P-based three-dimensional index for structural joins on distributed triple stores,” in *Proc. of DBISP2P Workshops*, 2006, pp. 323–330.

[26] M. Cai, M. R. Frank, B. Yan, and R. M. MacGregor, “A subscribable peer-to-peer RDF repository for distributed metadata management,” *Journal of Web Semantics*, vol. 2, no. 2, pp. 109–130, 2004.

[27] B. Quilitz and U. Leser, “Querying distributed RDF data sources with SPARQL,” in *Proc. of the European Semantic Web Conference (ESWC)*, 2008, pp. 524–538.

[28] H. Stuckenschmidt, R. Vdovjak, J. Broekstra, and G.-J. Houben, “Towards distributed processing of RDF path queries,” *International Journal of Web Engineering and Technology*, vol. 2, no. 2/3, pp. 207–230, 2005.

[29] N. Li, J. Rao, E. J. Shekita, and S. Tata, “Leveraging a scalable row store to build a distributed text index,” in *Proc. of the International CIKM Workshop on Cloud Data Management (CloudDb)*, 2009, pp. 29–36.

[30] H. Choi, J. Son, Y. Cho, M. K. Sung, and Y. D. Chung, “SPIDER: a system for scalable, parallel / distributed evaluation of large-scale RDF data,” in *Proc. of the International Conference on Information and Knowledge Management (CIKM)*, 2009, pp. 2087–2088.

[31] *HBase Graph for Jena*, http://cs.utdallas.edu/semanticweb/HBase-Extension/hbase-extension.html.

[32] *Third Provenance Challenge*, http://twiki.ipaw.info/bin/view/Challenge/ThirdProvenanceChallenge.

[33] *Open Provenance Model*, http://openprovenance.org.

[34] *SWAT Projects - the Lehigh University Benchmark (LUBM)*, http://swat.cse.lehigh.edu/projects/lubm/.

APPENDIX

APPENDIX

AGGREGATE RESULTS FOR PC3 AND LUBM

PC3 Aggregate Results						
Load Times (ms)	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**
HBase	416	2,245	7,292	59,495	672,144	7,380,597
MySQL Cluster	20	530	2,820	30,250	349,390	---
Q1 (ms)	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**
HBase	4	5	4	5	5	10
MySQL Cluster	9	9	9	9	9	---
Q2 (ms)	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**
HBase	4	5	4	9	9	27
MySQL Cluster	10	10	10	10	10	---
Q3 (ms)	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**
HBase	11	11	11	12	14	47
MySQL Cluster	11	11	11	11	11	---

LUBM Aggregate Results						
Load Times (ms)	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**
HBase	4,746	53,301	111,185	388,971	655,092	930,209
MySQL Cluster	1,540	24,970	55,610	196,110	341,580	492,960
Q1 (ms)	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**
HBase	12	11	13	11	11	14
MySQL Cluster	39	1,680	3,628	12,001	21,060	29,342
Q2 (ms)	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**
HBase	12	11	13	115,814	179,206	266,008
MySQL Cluster	129	9,574	21,985	91,881	180,557	290,882
Q3 (ms)	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**
HBase	12	13	13	13	15	16
MySQL Cluster	83	5,330	11,746	39,766	69,290	98,824
Q4 (ms)	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**
HBase	137	141	141	148	121	147
MySQL Cluster	50	51	52	52	52	52
Q5 (ms)	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**
HBase	632	640	650	643	643	669
MySQL Cluster	133	132	133	133	133	133
Q6 (ms)	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**
HBase	467	469	488	475	463	455
MySQL Cluster	199	193	194	199	197	194
Q7 (ms)	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**
HBase	77	80	115	120	114	113
MySQL Cluster	62	734	1,556	5,051	8,574	12,048

LUBM Aggregate Results						
Q8 (ms)	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**
HBase	8,591	9,225	8,880	9,951	9,783	9,783
MySQL Cluster	4,220	4,311	4,340	4,454	4,555	4,628
Q9 (ms)	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**
HBase	1,050	1,106	1,079	1,128	1,117	1,115
MySQL Cluster	4,066	4,121	4,149	4,192	4,231	4,233
Q10 (ms)	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**
HBase	11	14	11	11	11	11
MySQL Cluster	1,264	1,278	1,268	1,283	1,284	1,296
Q11 (ms)	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**
HBase	222	235	226	235	250	218
MySQL Cluster	49	228	457	1,450	2,421	3,421
Q12 (ms)	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**
HBase	277	297	282	284	314	291
MySQL Cluster	31	43	58	123	188	248
Q13 (ms)	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**
HBase	6	6	6	8	8	18
MySQL Cluster	122	122	121	122	123	126
Q14 (ms)	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**
HBase	380	1,162	1,544	2,604	5,088	6,185
MySQL Cluster	162	541	949	4,241	8,060	12,433

LUBM Aggregate Results						
Load Times (ms)	**L7**	**L8**	**L9**	**L10**	**L11**	
HBase	1,182,541	1,472,860	2,805,965	5,883,081	9,748,935	
MySQL Cluster	813,040	1,037,760	---	---	---	
Q1 (ms)	**L7**	**L8**	**L9**	**L10**	**L11**	
HBase	15	13	15	44	27	
MySQL Cluster	38,258	47,212	---	---	---	
Q2 (ms)	**L7**	**L8**	**L9**	**L10**	**L11**	
HBase	379,266	484,783	---	---	---	
MySQL Cluster	420,029	567,855	---	---	---	
Q3 (ms)	**L7**	**L8**	**L9**	**L10**	**L11**	
HBase	16	16	17	31	28	
MySQL Cluster	132,931	163,357	---	---	---	
Q4 (ms)	**L7**	**L8**	**L9**	**L10**	**L11**	
HBase	150	152	133	151	153	
MySQL Cluster	52	53	---	---	---	
Q5 (ms)	**L7**	**L8**	**L9**	**L10**	**L11**	
HBase	676	669	666	670	645	
MySQL Cluster	133	133	---	---	---	
Q6 (ms)	**L7**	**L8**	**L9**	**L10**	**L11**	
HBase	482	497	470	501	470	
MySQL Cluster	196	197	---	---	---	
Q7 (ms)	**L7**	**L8**	**L9**	**L10**	**L11**	
HBase	81	108	106	124	123	
MySQL Cluster	15,462	19,089	---	---	---	

LUBM Aggregate Results						
Q8 (ms)	**L7**	**L8**	**L9**	**L10**	**L11**	
HBase	9,795	9,855	9,563	9,801	10,099	
MySQL Cluster	4,674	4,746	---	---	---	
Q9 (ms)	**L7**	**L8**	**L9**	**L10**	**L11**	
HBase	1,182	1,230	1,206	1,310	1,561	
MySQL Cluster	4,236	4,222	---	---	---	
Q10 (ms)	**L7**	**L8**	**L9**	**L10**	**L11**	
HBase	13	15	14	26	27	
MySQL Cluster	1,282	1,282	---	---	---	
Q11 (ms)	**L7**	**L8**	**L9**	**L10**	**L11**	
HBase	278	320	281	302	463	
MySQL Cluster	4,403	5,422	---	---	---	
Q12 (ms)	**L7**	**L8**	**L9**	**L10**	**L11**	
HBase	341	406	384	474	702	
MySQL Cluster	310	374	---	---	---	
Q13 (ms)	**L7**	**L8**	**L9**	**L10**	**L11**	
HBase	22	20	19	17	16	
MySQL Cluster	123	123	---	---	---	
Q14 (ms)	**L7**	**L8**	**L9**	**L10**	**L11**	
HBase	9,939	16,362	---	---	---	
MySQL Cluster	22,263	26,456	---	---	---	

BIOGRAPHICAL SKETCH

Craig M. Franke earned a Master of Science in Computer Science from the University of Texas – Pan American in May 2011. He earned an Associate of Applied Science in Computer Science from Texas State Technical College (TSTC) – Harlingen in 2001 and a Bachelor of Applied Technology in Computer Information Systems from the University of Texas – Brownsville in 2008. He has spent close to 10 years with TSTC's Network and Telecommunication Services department in multiple roles, most recently as Assistant Director of Network Services. His roles have covered numerous technology areas from web development to system integration to network and data center operations. His permanent mailing address is 905 North Loop 499 Apt 212, Harlingen, TX, 78550.