8-2010

Distributed storage and queryng techniques for a semantic web of scientific workflow provenance

Jaime Alberto Navarro
University of Texas-Pan American

DISTRIBUTED STORAGE AND QUERYING TECHNIQUES FOR A SEMANTIC WEB OF SCIENTIFIC WORKFLOW PROVENANCE

A Thesis

by

JAIME ALBERTO NAVARRO

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

August 2010

Major Subject: Computer Science

DISTRIBUTED STORAGE AND QUERYING TECHNIQUES FOR A SEMANTIC WEB OF SCIENTIFIC WORKFLOW PROVENANCE

A Thesis

by

JAIME ALBERTO NAVARRO

COMMITTEE MEMBERS

Dr. Artem Chebotko
Chair of Committee

Dr. Richard Fowler
Committee Member

Dr. Zhixiang Chen
Committee Member

Dr. John Abraham
Committee Member

August 2010

ABSTRACT

Navarro, Jaime A., Distributed Storage and Querying Techniques for a Semantic Web of Scientific Workflow Provenance. Master of Science (MS), August, 2010, 41 pp., 4 tables, 6 figures, 41 references, 13 titles and 1 appendix.

In scientific workflow environments, scientists depend on provenance, which records the history of an experiment. Resource Description Framework is frequently used to represent provenance based on vocabularies such as the Open Provenance Model. For complex scientific workflows that generate large amounts of RDF triples, single-machine provenance management becomes inadequate over time. In this thesis, we research how HBase capabilities can be leveraged for distributed storage and querying of provenance data represented in RDF. We architect the *ProvBase* system that incorporates an HBase/Hadoop backend, propose a storage schema to hold provenance triples, and design querying algorithms to evaluate SPARQL queries in the system. We conduct an experimental study to show the feasibility of our approach.

DEDICATION

To my parents and sister

ACKNOWLEDGMENTS

I would like to thank my parents, Jaime Alfonso Navarro and Ervia Rosario Yerena, for their support; they taught me that everything is possible with hard work and determination. I also thank my wonderful sister, Lariza Anaiz Navarro, for all her advice and encouragement to graduate with a Thesis, and all the special people who were there for me.

I would especially like to thank Dr. Artem Chebotko for his support and encouragement during the process. I would also like to thank Tony Piazza for his invaluable help organizing and configuring the experiments. Finally, I would like to thank the Computer Science Department for all the support in making this thesis possible.

TABLE OF CONTENTS

ABSTRACT iii

DEDICATION iv

ACKNOWLEDGMENTS v

TABLE OF CONTENTS v

LIST OF TABLES viii

LIST OF FIGURES ix

CHAPTER I. INTRODUCTION 1

CHAPTER II. RELATED WORK 4

CHAPTER III. SYSTEM ARCHITECTURE 6

CHAPTER IV. STORAGE SCHEMA AND DATA INGEST 9

CHAPTER V. QUERYING ALGORITHMS 12

CHAPTER VI. PERFORMANCE STUDY 18

- Experimental Setup 18
- Datasets and Queries 19
- Data Ingest Performance 20
- Query Evaluation Performance 21

Query Optimization Discussion....21

CHAPTER VII. RESULTS AND DISCUSSION 24

REFERENCES 25

BIOGRAPHICAL SKETCH 41

LIST OF TABLES

Table 1: Datasets of Experiments 19

Table 2: Test Queries 20

Table 3: Data Ingest Performance 21

Table 4: Query Performances. 22

LIST OF FIGURES

Figure 1: The *ProvBase* Architecture. 8

Figure 2: A Table Row Structure. 9

Figure 3: Provenance Storage Schema and Instance. 11

Figure 4: Algorithm 1 Matching a Triple Pattern Over Triple 13

Figure 5: Algorithm 2 Matching a Triple Pattern Over a Database. 15

Figure 6: Algorithm 3 Matching a Basic Graph Pattern Over a Database 17

CHAPTER I

INTRODUCTION

Provenance has been identified as an important requirement for scientific workflows [1], [2], [3]. Scientific workflow management systems (SWfMS) [4], [5], [6], [7], [8], [9], [10] support design and execution of scientific workflows, as well as collection of provenance data to enable scientific discovery reproducibility, result interpretation, and problem diagnosis in insilico experiments. With scientific workflows emerging as a powerful paradigm for formalizing and automating complex and data intensive scientific processes, the role of provenance data management cannot be underestimated. E-scientists are now enabled to efficiently execute scientific workflows numerous times with different settings, parameters, and inputs in their continuous search for interesting results. In this scenario, large amounts of provenance data, which records the history of all workflow executions, are produced. Moreover, recent efforts of the community on provenance interoperability [11], [12] in different SWfMSs further suggest that provenance data from various scientific projects can be integrated on a global scale into a Semantic Web of Scientific Workflow Provenance to enable data analysis and provenance applications that span over multiple projects and organizations. Hence, there is a growing need for efficient database systems that can employ distributed storage and querying techniques to cope with large-scale provenance data management.

Inspired by Google's Bigtable [13] that successfully caches most data on the Web, in this paper, we examine how this powerful technology can be adapted to store and query scientific

workflow provenance data represented using Resource Description Framework (RDF). For this purpose, we choose to use HBase [14], an open-source implementation of Bigtable that is designed for massive scalability, which is particularly important for storage of large provenance datasets. HBase is not a relational database and does not support a high-level query language like SQL, but it provides an API for performing data updates and executing queries. HBase builds on top of Hadoop [15], a Java framework that supports intensive data communication among computers in a cluster. Hadoop has the capability to connect and coordinate thousands of nodes inside a cluster. It takes into account the geography of clusters, the location of the information and its proximity to other clusters, the continuity and the duplicity of data, and determines how to distribute data to obtain the best performance. Hadoop applications, such as HBase, utilize these capabilities without the need to interact with a file system spread among many nodes in a cluster. The communities of both Hadoop and HBase are growing and active. This means we can be rather confident that continued development and support will be available.

There are a number of questions that we need to answer when using HBase for provenance data management. What database schema is suitable for storing RDF triples to efficiently support triple pattern matching? How can SPARQL queries be evaluated in the database as HBase's API only supports simple retrieval by a table row key? This thesis reports our first attempt to address these issues. We architect the *ProvBase* system that incorporates an HBase/Hadoop backend for distributed storage and querying of provenance data. The main contributions of our work include a three-table storage schema that can be instantiated in HBase to hold provenance triples and querying algorithms that evaluate SPARQL queries in HBase using its native API. Using the Third Provenance Challenge queries, we conduct an experimental study to show the feasibility of *ProvBase*. The organization of this thesis is as follows. Related

work is discussed in Chapter II. The architecture of *ProvBase* is described in Chapter III. Our storage schema and data ingest are introduced in Chapter IV. Querying algorithms are presented in Chapter V. Finally, our performance study and concluding remarks are reported in Chapters VI and VII, respectively.

CHAPTER II

RELATED WORK

The scientific workflow community has developed a number of solutions for provenance data collection and management. We briefly summarize some of them in the following. VIEW [4] has an OWL vocabulary for provenance collection and develops a relational RDF store, called RDFProv [16], to store and query provenance using semantic web and relational technologies. Kepler [6] uses a provenance framework, called Collection-Oriented Modeling and Design (COMAD) [17], [18], that stores provenance data in an XML file and, recently, in a relational database [19], [20]. Taverna [5], [21], [22], [23] features a provenance ontology and employs semantic web technologies for provenance collection and representation. VisTrails [7], [24], [25] and Karma [26], [27], [28] use XML and relational database technologies to represent, store, and query workflow provenance. Trident [29], [30] and PreServ/PASOA [31], [32] are not tied to any particular provenance storage model, but rather define interfaces and adaptors to enable different storage systems, such as a file system, relational database, XML database and so forth. Finally, Swift [10], [33] implements a Virtual Data System (VDS) consisting of a set of relations to manage provenance information in a relational database. While all of these systems use proprietary mechanisms for modeling, collecting, and storing provenance, some also add support for the Open Provenance Model (OPM) [12], an emerging community-driven standard that facilitates provenance interoperability [11].

In terms of RDF data management, the projects most related to our work are Heart [34] and SPIDER [35]. These recently started projects use Hadoop [15] and HBase [14] to manage large RDF datasets; at the time of writing this thesis, there was not much information released to the public about these systems. Some other related research works on distributed RDF querying include [36], [37], [38], [39], [40]. In [36], a schema to store RDF data in the Hadoop file system and querying algorithms to evaluate SPARQL basic graph patterns are proposed. While [36] and this work address similar data management problems, the two approaches are different in their use of Hadoop and HBase, respectively. RDFCube [37] and RDFPeers [38] focus on RDF query processing in peer-to-peer environments. [39] and [40] study mediation techniques for federated querying of distributed RDF sources.

CHAPTER III

SYSTEM ARCHITECTURE

The *ProvBase* architecture is shown in Figure 1. It consists of four primary components: (1) clients that collect and query provenance, (2) *ProvBase* servers that process all client requests, (3) one active HBase master server that coordinates an HBase cluster, and (4) HBase region servers that store provenance. Clients, which are typically represented by scientific workflow management systems, collect provenance data to be stored in RDF format and execute SPARQL queries to retrieve stored provenance data. Clients send requests to *ProvBase* servers using a standard web services interface. One or more identical *ProvBase* servers deployed in a cluster handle client requests for provenance insertion and querying. Each *ProvBase* server implements algorithms for creation, population, and querying HBase tables. An HBase installation consists of one active HBase master server that may have inactive replicas and one or more HBase region servers that store provenance data. *ProvBase* servers communicate with both master and region servers.

One of the most important responsibilities of the master server is the creation and assignment of regions inside each region server. Each region assigned by the master server corresponds to a subset of data that a region server can hold, such that data is distributed among region servers in a cluster. This design allows massive scalability. The master server also monitors the existence and reachability of region servers, balances their workload, and handles

schema administration. The master server intercepts the initial administrative request from *ProvBase* servers, not to service the request, but to assist in identifying the correct region server to satisfy the request. If the active master server is down and there are no master replicas to take its duties, the entire HBase system goes offline. Unlike the master server, it is possible for some region servers to be taken offline, because data that each region server holds is replicated across other region servers.

Region servers handle all insertion (write) and querying (read) requests from *ProvBase* servers. When a write request is sent to a region server, the request is written in a region log and subsequently in an HBase table, distributed file system and, at the lowest level, Hadoop data nodes. In addition, cache may maintain a copy of data. When a read request is submitted, cache is checked first and if requested triples are not found, an HBase table is searched. A response to the read request is sent directly from a region server to a *ProvBase* server, which further evaluates SPARQL operators over received triples according to a client query and sends the final result to the client.

Additional details about the internal structure of HBase master and region servers are available in [14]. In the following sections, our focus is on a *ProvBase* server design, including a storage schema that it creates in HBase, data ingest and querying algorithms.

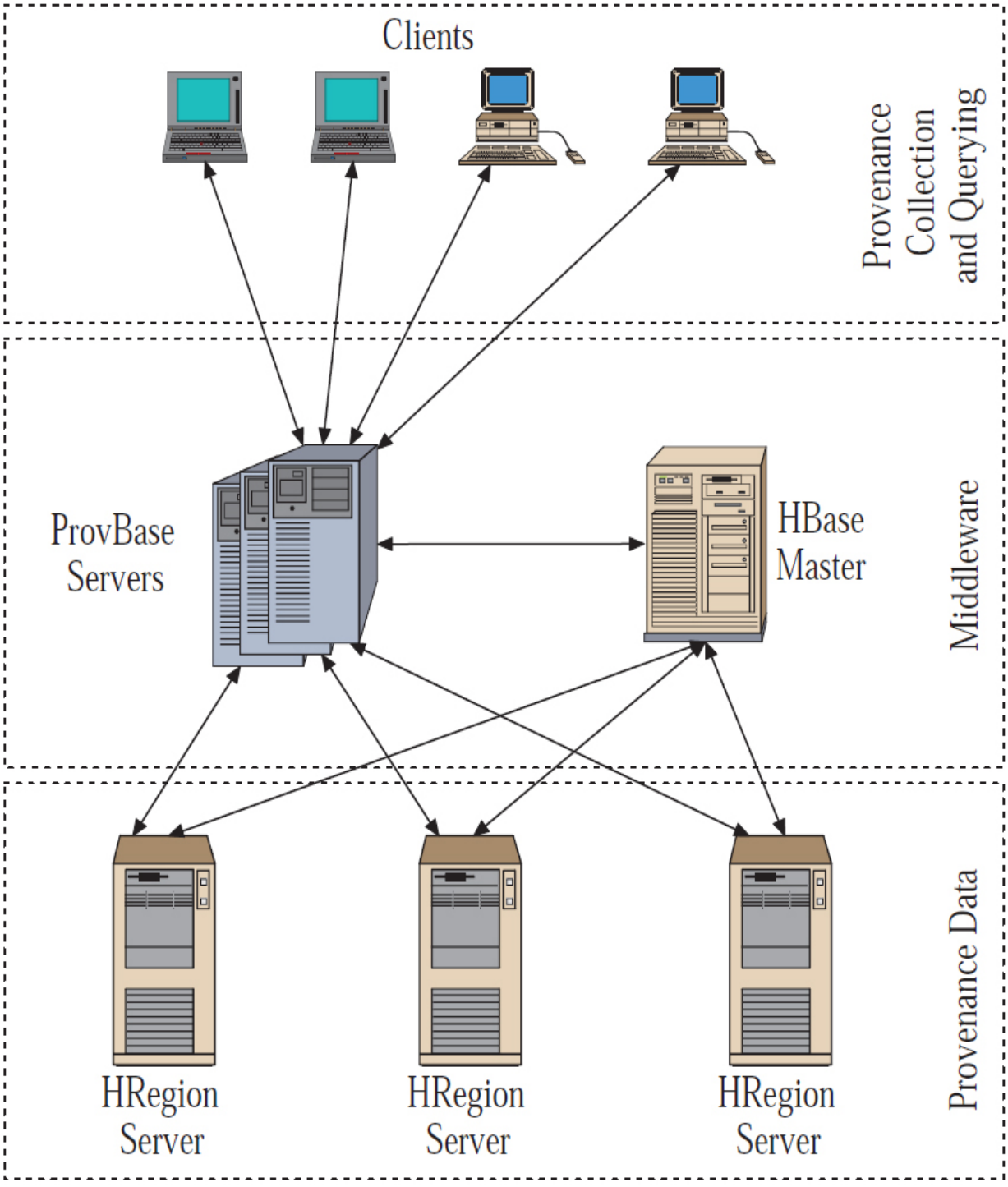

Figure 1: The *ProvBase* Architecture.

CHAPTER IV

STORAGE SCHEMA AND DATA INGEST

HBase stores data in tables that are structurally different from relations used in traditional relational databases. An HBase table (hereafter "table" for short) has columns and rows. While conceptually a table is a single entity, physically its columns are stored in separate data structures and its rows can be further partitioned horizontally into regions and distributed over region servers. Each row has a row key and a cell set as schematically shown in Figure 2. A row key uniquely identifies a row and can be used for efficient retrieval of a row cell set. Each cell contains a data value and a time stamp; the latter can provide an additional benefit to provenance data to record time when a provenance triple is stored. This structure allows having several values in different cells for the same column and row key.

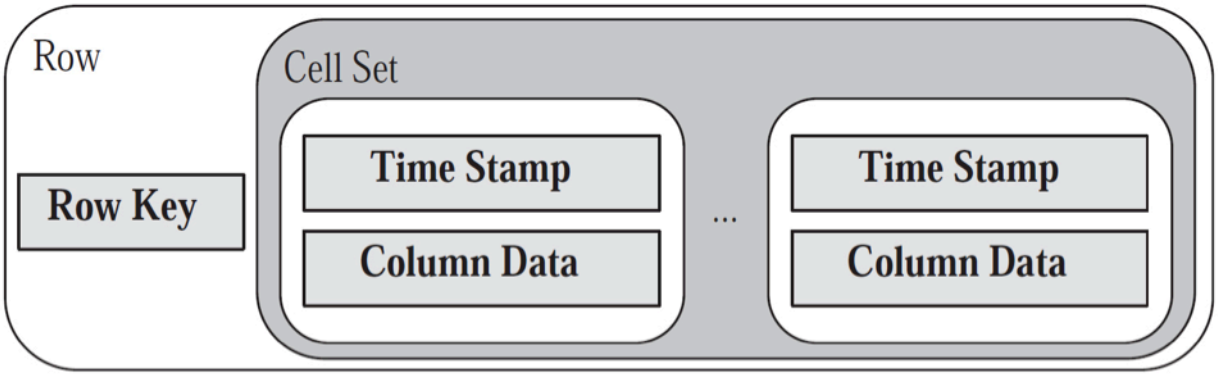

Figure 2: A Table Row Structure.

To store provenance data represented by a set of RDF triples, we propose using three tables with one column in each. The schemas and instances of these tables are depicted in Figure 3 (time stamps are omitted for brevity). Given the following sample triples

<D> <generatedArtifact> <A> .

<D> <generatedByProcess> <P> .

<C> <usedByProcess> <P> .

table T_s stores their subjects as row keys, and predicates and objects as values in column *po*, where a delimiter "*j*" denotes the boundary between terms. As shown in the figure, these triples are stored in two rows of table T_s, where the first row contains two cells in column *po*. Similarly, tables T_p and T_o store triple predicates and objects, respectively, as row keys and the other terms as column values.

This storage schema is intended for efficient matching of SPARQL triple patterns. For example, if a triple pattern contains a URI in the subject position, table T_s can be used to efficiently retrieve all predicates and objects stored in a row with this URI key. If a subject pattern is a variable, but a predicate or object pattern is not, then T_p or T_o can be used for retrieval. In case of a triple pattern with all variables, any table can be used to retrieve all provenance data in the system. An additional optimization can be used to improve time and space efficiency. For long URIs and literals that are to be stored as row keys, their hash values can be computed and stored instead. When triples need to be retrieved based on a long URI/literal key, a hash value is computed and used for retrieval. This optimization can be used on two arbitrary tables of the three proposed ones, since one table should store all original triple terms to support a retrieval of all stored data. Depending on a particular dataset, two tables can be optimized using this approach with T_o being a strong candidate since literals at object

positions can include sentences and paragraphs of text. Moreover, using this optimization, it becomes feasible to use additional tables which combine two (or even three) triple terms into a row key to enable even faster matching of triple patterns with one variable or no variables at all. A *ProvBase* server creates the three tables in the Hbase database during the system setup.

Data insertion is initiated by a client when it sends a set of triples or a URL of an RDF document to a *ProvBase* server. At that point, triples are converted to rows and fed to HBase, which further distributes them among region servers. An algorithm for data ingest is not complicated, such that three rows are created for each supplied triple and inserted into the corresponding tables. An implementation of the described optimization strategy that hashes URIs and literals whose lengths exceed a given threshold is also straightforward.

T_s

rk (Row Key)	po
<D>	<generatedArtifact> | <A> <generatedByProcess> | <P>
<C>	<usedByProcess> | <P>

T_p

rk (Row Key)	so
<generatedArtifact>	<D> | <A>
<generatedByProcess>	<D> | <P>
<usedByProcess>	<C> | <P>

T_o

rk (Row Key)	sp
<A>	<D> | <generatedArtifact>
<P>	<D> | <generatedByProcess> <C> | <usedByProcess>

Figure 3: Provenance Storage Schema and Instance.

CHAPTER V

QUERYING ALGORITHMS

In this section, we propose three algorithms that allow us to evaluate SPARQL queries over our HBase storage schema for provenance data. HBase supports simple querying mechanisms, such as retrieving all rows from a table and retrieving all values in some column for a row with a known key. Retrieved rows and values can be further processed via the returned iterators. Evaluation of more complex querying constructs, such as SPARQL triple patterns and basic graph patterns, which are needed to express provenance queries, requires additional research

Our first function, *matchTP-T*, allows matching of a triple pattern over a triple and is outlined in Algorithm1. It is a general-purpose function that depends on neither our storage schema nor HBase. *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.

Function *matchTP-DB* as outlined in Algorithm 2 is used to match a triple pattern *tp* in an HBase database *DB* according to our storage schema.

```
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 1 Matching a Triple Pattern Over Triple

The output of this function is a relation *R* that holds a set of all matching triples in the database. To match *tp* in *DB*, *matchTP-DB* may have several options. If *tp*'s subject pattern is not a variable, the function can efficiently access all values in column *po* of table *Ts* for the row with key *tp.sp* via an iterator *i*. A triple *t* = (*tp.sp, p, o*), where predicate *p* and object *o* are obtained by splitting a value in column *po*, is then matched with *tp* using function *matchTP-T*. If the match is successful, the triple is added to relation *R*. When *tp*'s subject pattern is a variable, matching using a row key in T_S is not possible. Hence, the function implements a similar strategy to match *tp* over tables T_O and T_P. It should be noted that T_P is the least recommended of these three options, since heuristically the corresponding iterator for this table may contain a substantially larger set of triples than the iterators for T_S and T_O. To summarize, retrieving triples based on a row key from T_P usually yields the highest selectivity. The final and most expensive case is when a triple pattern contains only variables. The result of matching this triple pattern

yields all triples in T_s, T_o, or T_p. This case is handled in the last loop of the algorithm. *matchTP-DB* obtains a scanner over all rows in T_s and, for each row, an iterator over all values in column *po*. Triples are reconstructed from a row key and a column value and added to relation *R* without calling *matchTP-T*. While strategies for retrieving all triples from T_o and T_p are similar (not shown in the algorithm), it is a good idea to randomly choose one of these three strategies to balance a workload.

Our final function *matchBGP-DB* is outlined in Algorithm 3. 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 set of tuples representing matched graphs. The idea behind *matchBGP-DB* is as follows. The algorithm first sorts triple patterns in the non-descending order of their selectivities, such that triple patterns that yield smaller results appear first in the list. This can potentially save some computation if matching triples do not fit into main memory. A simple approach to decide on a triple pattern selectivity is to use these heuristics: (1) if a triple pattern contains only variables, it has the highest selectivity, (2) if a triple pattern contains a non-variable only at the predicate pattern position, it has moderate selectivity, and (3) if a triple pattern has a non-variable at the subject and/or object pattern positions, it has a low selectivity. A formal and more advanced approach to triple pattern selectivity estimation is described in [41]. Next, each triple pattern is evaluated using the *matchTP-DB* function returning a relation with matching triples. These relations are joined according to their corresponding triple pattern order using a nested-loops-like join strategy.

1: **function** matchTP-DB
2: **input:** triple pattern $tp = (sp, pp, op)$, database $DB = \{T_s, T_p, T_o\}$
3: **output:** set of triples $R_{(sp,pp,op)} = \{t | t \text{ is in } DB \wedge t \text{ matches } tp\}$
4: $R = \emptyset$
5: **if** $tp.sp$ is not a variable **then**
6: Get an iterator i over all values in column po of table T_s for row with key $tp.sp$
7: **while** i.next() **do**
8: $t = (tp.sp, p, o)$, where $p \mid o = i.po$
9: **if** matchTP-T(tp, t) **then** $R = R \cup \{t\}$ **end if**
10: **end while**
11: **return** R
12: **end if**
13: **if** $tp.op$ is not a variable **then**
14: Get an iterator i over all values in column sp of table T_o for the row with key $tp.op$
15: **while** i.next() **do**
16: $t = (s, p, tp.op)$, where $s \mid p = i.sp$
17: **if** matchTP-T(tp, t) **then** $R = R \cup \{t\}$ **end if**
18: **end while**
19: **return** R
20: **end if**
21: **if** $tp.pp$ is not a variable **then**
22: Get an iterator i over all values in column so of table T_p for row with key $tp.pp$
23: **while** i.next() **do**
24: $t = (s, tp.pp, o)$, where $s \mid o = i.so$
25: **if** matchTP-T(tp, t) **then** $R = R \cup \{t\}$ **end if**
26: **end while**
27: **return** R
28: **end if**
29: Get an iterator/scanner ri over all rows in table T_s
30: **while** ri.next() **do**
31: Get an iterator i over all values in column po for row $ri.row$
32: **while** i.next() **do**
33: $t = (s, p, o)$, where $s = i.rk$ and $p \mid o = i.po$
34: $R = R \cup \{t\}$
35: **end while**
36: **end while**
37: **return** R
38: **end function**

Figure 5: Algorithm 2 Matching a Triple Pattern Over a Database.

An attempt is made to concatenate each triple from the first relation with every triple from the second relation based on a condition that the triples must agree on the values (bindings) of shared variables. Next, an attempt is made to join each concatenated tuple with triples from the next relation and so forth. Only those tuples that result from the concatenation of triples from every available relation are added to the final relation *R*. Function *matchTP-DB* does not materialize intermediate join results to reduce memory consumption, but rather processes all relations concurrently. Tuples that do not concatenate with triples in subsequent joins are quickly eliminated.

Other SPARQL features, such as projection (*SELECT*), filtering (*FILTER*), and alternative graph patterns (*UNION*), can be incorporated in the presented algorithmic framework in a straightforward fashion. Projection, which eliminates unwanted variables, is performed during triple pattern matching as long as to-be eliminated variables do not occur in other triple patterns of a query; in the latter case, such variables are projected out after a join phase. Similarly, filtering, which includes logical connectives ($\neg$, $\wedge$, $\vee$), inequality and equality operators ($<$, $\leq$, $\geq$, $>$), unary predicates (*bound*, *isIRI*), and other constructs, is performed during triple pattern matching if a filter expression only uses variables from one triple pattern or during basic graph pattern matching if the expression references variables from multiple triple patterns. A union of basic graph patterns corresponds to a union of relations that resulted from graph pattern matching. If relations are not union-compatible, their schemas must be extended accordingly before a union is computed [42]. Finally, dealing with optional graph patterns can be much more complicated due to non-trivial nesting and sequential occurrence of *OPTIONAL* constructs in a SPARQL query. Since the Third Provenance Challenge queries do not show the need for this construct, we leave out optional graph patterns for future work.

1: **function** matchBGP-DB
2: **input:** basic graph pattern $bgp = \{tp_1, tp_2, ..., tp_n\}$, database $DB = \{T_s, T_p, T_o\}$
3: **output:** set of tuples $R_{(tp_1.sp, tp_1.pp, tp_1.op, tp_2.sp, ...)} = \{g | g$ is a graph in $DB \wedge g$ matches $bgp\}$
4: $R = \emptyset$
5: Sort triple patterns of bgp in non-descending order of their estimated selectivities. Let ordered $bgp = (tp_1, tp_2, ..., tp_n)$
6: Get iterators $i_1, i_2, ..., i_n$ over triples in matchTP-DB(tp_1, DB), matchTP-DB(tp_2, DB), ..., matchTP-DB(tp_n, DB)
7: i_1.rewind()
8: **while** i_1.next() **do**
9: $t_1 = (i_1.s, i_1.p, i_1.o)$
10: i_2.rewind()
11: **while** i_2.next() **do**
12: **if** t_1 and $(i_2.s, i_2.p, i_2.o)$ agree on values of shared variables in tp_1 and tp_2 **then**
13: $t_2 = t_1 \bigoplus (i_2.s, i_2.p, i_2.o)$ /*concatenation*/
14: i_3.rewind()
15: **while** i_3.next() **do**
16: **if** t_2 and $(i_3.s, i_3.p, i_3.o)$ agree on values of shared variables in (tp_1, tp_2) and tp_3 **then**
17: $t_3 = t_2 \bigoplus (i_3.s, i_3.p, i_3.o)$
18: i_4.rewind()
19: **while** i_4.next() **do**
20: ...
21: **while** i_n.next() **do**
22: **if** t_{n-1} and $(i_n.s, i_n.p, i_n.o)$ agree on values of shared variables in $(tp_1, tp_2, ..., tp_{n-1})$ and tp_n **then**
23: $t_n = t_{n-1} \bigoplus (i_n.s, i_n.p, i_n.o)$
24: $R = R \cup t_n$
25: **end if**
26: **end while**
27: ...
28: **end while**
29: **end if**
30: **end while**
31: **end if**
32: **end while**
33: **end while**
34: **return** R
35: **end function**

Figure 6: Algorithm 3 Matching a Basic Graph Pattern Over a Database

CHAPTER VI

PERFORMANCE STUDY

We implemented our algorithms in Java and conducted performance experiments with *ProvBase*. *ProvBase* was used to load provenance datasets of varying sizes and execute test queries defined by the Third Provenance Challenge.

Experimental Setup

We used five identically configured Gateway E3600 computers for running our experiments. Each machine had a 1.8 GHz Pentium 4 processor, 1 GB RAM, 20 GB IDE hard drive and a gigabit Ethernet adapter. These machines were directly connected to a D-Link DGS-2208 gigabit switch and configured with static, non-routable IP addresses. These machines were all running version 5.0.3 of the Debian operating system, version 1.6.0 of the OpenJDK, version 0.20.1 of Hadoop and version 0.20.3 of HBase.

The Hbase/Hadoop installations on each machine were configured to use the fully distributed mode. The hard drives on each machine had at least 16 GB of free space. One of the machines was configured as the HBase master server and the other four machines were configured as HBase region servers. The *ProvBase* server code was executed on the same machine as the HBase master server.

Dataset	# of RDF triples	# of workflow runs	Disk space
$D1$	700	1	100KB
$D2$	7,000	10	1MB
$D3$	70,000	100	11MB
$D4$	700,000	1,000	111MB
$D5$	7,000,000	10,000	1.1GB
$D6$	70,000,000	100,000	11.4GB

Table 1: Datasets of Experiments

Datasets and Queries

The Third Provenance Challenge [11] employed the Load Workflow that was a variation of a workflow used in the Pan-STARRS project [43]. Via simulation, we generated a number of provenance documents for multiple runs of this workflow. Provenance data was represented using Tupelo's OWL vocabulary available from the Open Provenance Model website [12]. Each workflow execution generated approximately 700 RDF triples.

In our experiments, each dataset was characterized by the number of its constituent triples (say *n*), the number of involved workflow runs (*n/700*), and space that it occupied on a disk as shown in Table 1. In addition, we selected three provenance challenge questions and expressed them as SPARQL queries. Our queries are presented in Table 2 where *rdfs, opm,* and p refer to namespaces *http://www.w3.org/2000/01/rdf-schema#, http://www.ipaw.info/2007/opm#,* and *http://www.cs.panam.edu/provenance/,* respectively. The queries have varying complexity: *Q1* is the simplest query with one triple pattern, *Q2* has three triple patterns, and *Q3* is the most complex with six triple patterns.

The *FILTER* clause of each query only references one variable from the preceding triple pattern and checks whether a value of this variable contains a certain literal.

#	**Provenance question and SPARQL query**

$Q1$	Determine the step where halt occurred?
	`SELECT ?process WHERE {`
	`?process rdfs:label ?name .`
	`FILTER regex(?name, "halt")`
	`}`

$Q2$	Was the range check performed for a given table?
	`ASK {`
	`?table opm:generatedArtifact p:tableID .`
	`?table opm:generatedByProcess ?process .`
	`?process rdfs:label ?name .`
	`FILTER regex(?name, "IsMatchTableColumnRanges")`
	`}`

$Q3$	Which CSV files contributed to a given detection?
	`SELECT ?file WHERE {`
	`?detection opm:generatedArtifact p:detectionID.`
	`?detection opm:generatedByProcess ?process .`
	`?cause opm:usedByProcess ?process .`
	`?cause opm:usedArtifact ?file .`
	`?cause opm:usedRole ?role .`
	`?role rdfs:label ?name .`
	`FILTER regex(?name, "CSV")`
	`}`

Table 2: Test Queries

Data Ingest Performance

Data ingest performance for each dataset is reported in Table 3. The data loading performance revealed linear scalability for given dataset sizes. The average triple per second throughput (154tps) and time (4.6s) required to load provenance data from a single workflow run proved to be good for long-running, data and computationally intensive workflows.

Query Evaluation Performance

Test query performance for each dataset is reported in Table 4. The system showed good querying scalability with respect to the dataset size, i.e., the database growth by a factor of 10, resulted in the query response time growth by a factor of 2 – 3 on average. As the number of triple patterns in a query increased, the time to evaluate the query also increased. *Q1* and *Q3* were the fastest and slowest queries in our experiment, respectively.

Data Ingest Performance

Data-set	Loading time (s)	Triples per second	Time per work-flow run (s)
$D1$	5.817	120	5.817
$D2$	43.300	162	4.330
$D3$	409.873	171	4.098
$D4$	4077.898	172	4.077
$D5$	44952.232	156	4.495
$D6$	492330.444	142	4.923
	Average:	154	4.623

Table 3: Data Ingest Performance

Query Optimization Discussion

As we discussed previously, triple pattern evaluation is usually much slower over table T_p than over table T_s or table T_o, because there likely exist more triples with the same predicate rather than with the same subject or object. For example, in our provenance datasets, provenance of each workflow run contained almost 30 triples with predicate *opm:generatedArtifact*, which

were all stored in a single row of T_p. For the largest dataset with 100,000 workflow runs, the number of such triples approached 3 million.

On the other hand, the same dataset contained only one triple with object *p:tableID* that was stored into T_o. Therefore, the first triple pattern of $Q2$, *(?table, opm:generatedArtifact, p:tableID)*, was much more efficient when matched over table T_o. However, how could we match $Q2$'s second triple pattern, *(?table, opm:generatedByProcess, ?process)*, using T_s or T_o when only the predicate was known? To answer this question, we looked at this triple pattern in the context of the first triple pattern.

Data-set	**Query evaluation time (s)**		
	$Q1$	$Q2$	$Q3$
$D1$	0.034	0.035	0.147
$D2$	0.069	0.129	0.275
$D3$	0.124	0.235	0.446
$D4$	0.311	0.479	1.150
$D5$	0.879	1.399	3.870
$D6$	2.924	4.586	13.882

Table 4: Query Performances.

The latter matched one triple and thus had one binding for variable *?table*, say binding *p:table1*. Since variable *?table* should have had the same binding for every occurrence in $Q2$, we were able to replace the evaluation of *(?table, opm:generatedByProcess, ?process)* over T_p with the evaluation of *(p:table1, opm:generatedByProcess, ?process)* over T_s to obtain equivalent results. The same technique was used to optimize $Q2$'s third triple pattern, such that we were able to match it over T_s. We optimized $Q3$ in the same fashion to completely eliminate the need

to access T_p. As a result, optimized queries $Q2$ and $Q3$ showed nearly constant performance of around 0.032s and 0.054s, respectively, on every provenance dataset. The performance did not decrease with the growth of the dataset size, because only one workflow run produced a table and detection with given identifiers and therefore HBase was able to efficiently retrieve matching triples using its multi-level index structures.

CHAPTER VII

RESULTS AND DISCUSSION

The performance study showed promising results. We observed that *ProvBase*, deployed on a five-node cluster, was able to efficiently store and query provenance of up to 100,000 executions of a real life scientific workflow.

The additional optimization applied to queries *Q2* and *Q3* improved the query response times for the largest dataset by two and three orders of magnitude, respectively. We studied the problem of how scientific workflow provenance data represented in RDF can be stored and queried in a distributed environment. We proposed a storage schema for HBase that is tuned for efficient triple pattern matching and designed querying algorithms that execute SPARQL queries over an HBase provenance database. We implemented our algorithms in the *ProvBase* system and successfully evaluated their performance using the Third Provenance Challenge queries. *ProvBase*, deployed on a five-node cluster, showed promising results and was able to efficiently store and query provenance of up to 100, 000 executions of a real-life scientific workflow.Future work directions include adding support for optional graph pattern evaluation, distributing workload among *ProvBase* servers, and conducting performance studies on larger clusters with more powerful nodes. We also plan to further explore and conduct comprehensive evaluation of the discussed querying optimizations, including row key hashing, encoding multiple triple pattern terms or even basic graph patterns as row keys, and intelligent substitution of subsequent triple pattern terms in a query with already matched data.

REFERENCES

[1] Y. Simmhan, B. Plale, and D. Gannon, "A survey of data provenance in e-science," *SIGMOD Record*, vol. 34, no. 3, pp. 31–36, 2005.

[2] S. B. Davidson, S. C. Boulakia, A. Eyal, B. Ludäscher, T. M. McPhillips, S. Bowers, M. K. Anand, and J. Freire, "Provenance in scientific workflow systems," *IEEE Data Engineering Bulletin*, vol. 30, no. 4, pp. 44–50, 2007.

[3] S. B. Davidson and J. Freire, "Provenance and scientific workflows: challenges and opportunities," in *Proc. of the ACM SIGMOD International Conference on Management of Data*, 2008, pp. 1345–1350.

[4] C. Lin, S. Lu, X. Fei, A. Chebotko, D. Pai, Z. Lai, F. Fotouhi, and J. Hua, "A reference architecture for scientific workflow management systems and the VIEW SOA solution," *IEEE Transactions on Services Computing*, vol. 2, no. 1, pp. 79– 92, 2009.

[5] T. M. Oinn, R. M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. A. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. W. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe, "Taverna: lessons in creating a workflow environment for the life sciences," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1067–1100, 2006.

[6] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. B. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific workflow management and the Kepler system," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1039–1065, 2006.

[7] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "Managing the evolution of dataflows with VisTrails," in *Proc. of the International Conference on Data Engineering Workshops*, 2006, p. 71.

[8] J. Kim, E. Deelman, Y. Gil, G. Mehta, and V. Ratnakar, "Provenance trails in the Wings/Pegasus system," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, pp. 587–597, 2008.

[9] Y. Simmhan, R. Barga, C. van Ingen, E. Lazowska, and A. Szalay, "Building the Trident scientific workflow workbench for data management in the cloud," *Proc. of the International Conference on Advanced Engineering Computing and Applications in Sciences*, pp. 41–50, 2009.

[10] Y. Zhao, M. Hategan, B. Clifford, I. T. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde, "Swift: Fast, reliable, loosely coupled parallel computation," in *Proc. of the International Workshop on Scientific Workflows*, 2007, pp. 199–206.

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

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

[13] 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.

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

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

[16] A. Chebotko, X. Fei, C. Lin, S. Lu, and F. Fotouhi, "Storing and querying scientific workflow provenance metadata using an RDBMS," in *Proc. of the IEEE International Conference on e-Science*, 2007, pp. 611–618.

[17] S. Bowers, T. M. McPhillips, S. Riddle, M. K. Anand, and B. Ludäscher, "Kepler/pPOD: Scientific workflow and provenance support for assembling the tree of life," in *Proc. of the International Provenance and Annotation Workshop*, 2008, pp. 70–77.

[18] S. Bowers, T. M. McPhillips, M. Wu, and B. Ludäscher, "Project histories: Managing data provenance across collection-oriented scientific workflow runs," in *Proc. of the International Workshop on Data Integration in the Life Sciences*, 2007, pp. 122–138.

[19] M. K. Anand, S. Bowers, T. M. McPhillips, and B. Ludäscher, "Efficient provenance storage over nested data collections," in *Proc. of the International Conference on Extending Database Technology*, 2009, pp. 958–969.

[20] ——, "Exploring scientific workflow provenance using hybrid queries over nested data and lineage graphs," in *Proc. of the International Conference on Scientific and Statistical Database Management*, 2009, pp. 237–254.

[21] A. D. Preece, P. Missier, S. M. Embury, B. Jin, and R. M. Greenwood, "An ontology-based approach to handling information quality in e-science," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 3, pp. 253–264, 2008.

[22] P. Missier, K. Belhajjame, J. Zhao, M. Roos, and C. A. Goble, "Data lineage model for Taverna workflows with lightweight annotation requirements," in *Proc. of the International Provenance and Annotation Workshop*, 2008, pp. 17–30.

[23] J. Zhao, C. A. Goble, R. Stevens, and D. Turi, "Mining Taverna's semantic web of provenance," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, pp. 463–472, 2008.

[24] J. Freire, C. T. Silva, S. P. Callahan, E. Santos, C. E. Scheidegger, and H. T. Vo, "Managing rapidly-evolving scientific workflows," in *Proc. of the International Provenance and Annotation Workshop*, 2006, pp. 10–18.

[25] C. T. Silva, J. Freire, and S. P. Callahan, "Provenance for visualizations: Reproducibility and beyond," *Computing in Science and Engineering*, vol. 9, no. 5, pp. 82–89, 2007.

[26] Y. L. Simmhan, B. Plale, and D. Gannon, "A framework for collecting provenance in data-centric scientific workflows," in *Proc. of the International Conference on Web Services*, 2006, pp. 427–436.

[27] ——, "Karma2: Provenance management for data-driven workflows," *International Journal of Web Services Research*, vol. 5, no. 2, pp. 1–22, 2008.

[28] B. Cao, B. Plale, G. H. Subramanian, E. Robertson, and Y. L. Simmhan, "Provenance information model of karma version 3," in *Proc. of the International Workshop on Scientific Workflows*, 2009, pp. 348–351.

[29] M. D. Valerio, S. S. Satya, R. S. Barga, and J. J. Jared, "Capturing workflow event data for monitoring, performance analysis, and management of scientific workflows," in *Proc. of the IEEE International Conference on e-Science*, 2008, pp. 626–633.

[30] Y. L. Simmhan, R. Barga, and C. van Ingen, "Automatic provenance recording for scientific data using Trident," in *American Geophysical Union (AGU) Fall Meeting*, 2008.

[31] L. Moreau, P. T. Groth, S. Miles, J. V´azquez-Salceda, J. Ibbotson, S. Jiang, S. Munroe, O. F. Rana, A. Schreiber, V. Tan, and L. Z. Varga, "The provenance of electronic data," *Communications of the ACM*, vol. 51, no. 4, pp. 52–58, 2008.

[32] P. T. Groth and L. Moreau, "Recording process documentation for provenance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 9, pp. 1246–1259, 2009.

[33] Z. Hou, M. Wilde, M. Hategan, X. Zhou, I. T. Foster, and B. Clifford, "Experiences of on-demand execution for large scale parameter sweep applications on OSG by Swift," in *Proc. of the International Conference on High Performance Computing and Communications*, 2009, pp. 527–532.

[34] *Heart*, http://heart.korea.ac.kr.

[35] 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 ACM Conference on Information and Knowledge Management*, 2009, pp. 2087– 2088.

[36] M. F. Husain, P. Doshi, L. Khan, and B. M. Thuraisingham, "Storage and retrieval of large RDF graph using Hadoop and MapReduce," in *Proc. of the International Conference on Cloud Computing*, 2009, pp. 680–686.

[37] 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 the Databases, Information Systems, and Peer-to-Peer Computing, International Workshops*, 2006, pp. 323–330.

[38] 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.

[39] B. Quilitz and U. Leser, "Querying distributed RDF data sources with SPARQL," in *Proc. of the European Semantic Web Conference*, 2008, pp. 524–538.

[40] 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.

[41] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds, “SPARQL basic graph pattern optimization using selectivity estimation,” in *Proc. of the International Conference on World Wide Web*, 2008, pp. 595–604.

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

[43] Y. Simmhan, R. S. Barga, C. van Ingen, M. A. Nieto- Santisteban, L. Dobos, N. Li, M. Shipway, A. S. Szalay, S. Werner, and J. Heasley, “GrayWulf: Scalable software architecture for data intensive computing,” in *Proc. of the Hawaii International Conference on System Sciences*, 2009, pp. 1–10.

APPENDIX A

APPENDIX A

PROVBASE INSTALLATION MANUAL

In order to complete the installation it is assumed the following elements are present:

4 or more computers with similar hardware characteristics (hardware optional)

Linux Debian Distribution Release X.X.X

A network installation/Configuration that connects all nodes

Internet Access of at least one node to download the latest versions of files (not necessary)

System Installation

To install *ProvBase*, it is necessary to obtain the latest versions of JAVA, HADOOP and HBASE. The following steps explain in detail how to acquire and properly install the required programs.

First, in order to install Java, the application installer "apt-get" provided by Debian will be used. In the command prompt, the following command must be typed:

apt-get update

This will update the repositories of the OS installer. The next command:

apt-get install sun-java6-jdk

Will install Java in the System; JAVA/BIN needs to be installed in the current path. The aforementioned can be verified by typing:

echo $PATH

It is important to work with the latest version of Hadoop and HBase. For this purpose, links are provided for both of these programs. The links might change in the future; therefore, it is convenient to revise each project website, and make sure to download the latest version of them. This manual is using Hadoop Release xxx, and HBase release xxx.

Download Hadoop:

wget http://mirror.cloudera.com/apache/hadoop/core/hadoop-0.X.X/hadoop-0.X.X.tar.gz

Download HBase:

wget http://people.apache.org/~stack/hbase-0.X.X-stable/hbase-0.X.X.tar.gz

After the program files are downloaded, a folder called *ProvBase* should be created inside the root folder and the downloaded files placed inside it. The contents of these files need to be extracted into individual folders named according to the project. The final folder structure should look like this: */ProvBase/hbase/ /ProvBase/hadoop/*

The next step is to copy the source of *ProvBase* to a folder named src inside the *ProvBase* folder as follows, */ProvBase/src/*

One must make sure to include all the Java files and scripts.

SSH Configuration

The following steps should only be done for the main node. Later in this manual, it will be explained how to copy this configuration to the rest of the nodes in the cluster. This installation assumes the main node is the future "Master Server" (review the *ProvBase* configurations scheme for further explanation).

In order to run HBase and Hadoop it is necessary to install and configure SSH servers and clients. This will also help us to expedite the whole installation.

It should be verified that the SSH client/server is installed in the system; this can be done by typing:

ssh–version

If the installation was done correctly, one can continue to the next step. However, if the system can't find the program, by typing:

apt-get ssh

One can make sure all the nodes in the cluster have SSH installed.

The next step is to allow the master node (the current node) to access the other nodes; this step is necessary in order to make multiple and concurrent installations, and will later permit the communication between the nodes.

To accomplish this, the code below will create a public/private key and save it in a file name: *authorized_keys.*

ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa

cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys

It is necessary to copy this file to each node in the cluster to gain access without being requested for the password every time. The file must be saved in the folder *./ssh/authorized_keys* of each node.

Once the file is in all the nodes, the connection to every node should be verified. This can be done with the following command, where $node is the ip address or the name of the node. ssh $node

Note that the username in each node needs to be the same for all the nodes.

Cluster Configuration

The following section describes how to transfer the files to all the nodes.

To begin, it is necessary to create a file named servers and add the ip addresses or name of each node of the cluster one per line, for instance:

192.1.2.1

192.1.2.2

192.1.2.3

The following script will generate the correct environment in each node. The script will copy the source code from the main node and transfer it over to the rest of nodes. (The script assumes the compressed files downloaded from HBase and Hadoop project are named: *hbase.tar.gz* and *hadoop.tar.gz*)

for host in `cat servers` do echo $host

ssh $host 'apt-get update; apt-get upgrade; apt-get install sun-java6-jdk;

mkdir –p /ProvBase'

scp /ProvBase/hadoop.tar.gz /provbae/hbase.tar.gz $host:~

ssh $host 'mkdir -p /ProvBase/pkgs; cd /ProvBase/pkgs; tar xzf ~/hadoop.tar.gz; tar xzf ~/hbase.tar.gz; ln -s /ProvBase/pkgs/hadoop /ProvBase/hadoop; ln –s /ProvBase/pkgs/hbase /ProvBase/hbase'

done

The script will then connect to each node and update the repository, install Java and create the folder *ProvBase*. Then, it will copy the tar files from hadoop and hbase to the node. Finally, it will expand each project file and save their contents in their respective folders, exactly as it is in the main node.

System Configuration

Checking that Hadoop and HBase environment paths are correctly configured with the system paths is a must. It is also important to include in the Hadoop configuration the HBase path and vice versa.

export JAVA_HOME= usr/lib/java

export HADOOP_CLASSPATH=/*ProvBase*/hbase/hbase

0.X.X.jar:/*ProvBase*/hbase/hbase-0.X.X-test.jar::/*ProvBase*/hadoop/hadoop-0.X.X.jar:/conf

export HBASE_CLASSPATH=/*ProvBase*/hadoop/conf

export HBASE_MANAGES_ZK=true

Hadoop Configuration

The configuration of Hadoop, involves modifying several files, most of them XML settings.

In the folder Hadoop there is a subfolder named conf that contains a file named "*core-site.xml*".

One needs to erase its content and copy the code below:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>
  <property>
     <name>hadoop.tmp.dir</name>
     <value>/provbase/hadoop</value>
  </property>
  <property>
     <name>fs.default.name</name>
     <value>hdfs://node1.local:50001</value>
  </property>
  <property>
     <name>tasktracker.http.threads</name>
     <value>80</value>
   </property>
</configuration>
```

The next file that needs to be updated is *mapred-site.xml.* It can be found in the same conf folder.

Its content must be erased and substituted with the following code:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>  <configuration>

  <property>
     <name>mapred.job.tracker</name> <value>nod1.local:50002</value>
  </property>

  <property>
        <name>mapred.tasktracker.map.tasks.maximum</name> <value>4</value>
  </property>

  <property>
        <name>mapred.tasktracker.reduce.tasks.maximum</name> <value>4</value>
  </property>

  <property>
        <name>mapred.output.compress</name> <value>true</value>
  </property>

  <property>
        <name>mapred.output.compression.type</name> <value>BLOCK</value>
  </property>

</configuration>
```

The last file to modify is *hdfs-site.xml*. The only attribute that is required to modify is the *dfs.replication*. Set the same number of nodes in the cluster.

```
<property>
      <name>dfs.replication</name>
      <value>3</value>
</property>
```

Inside of the folder conf create two files named, Masters and Slaves. In the Master file, add the master node IP or the qualified domain (node1.local)

In Slaves, add the list of ip address for each node, one per line.

The last step consist in copying over the all the configuration files to the rest of the nodes. This can be accomplished by running the script below. It will copy over the new configurations files to each node and format each namenode space, including the master node.

```
for host in `cat slaves` do
  echo $host
  scp slaves masters hdfs-site.xml hadoop-env.sh core-site.xml ${host}:/ProvBase/hadoop/conf
  ssh $host '/mnt/hadoop/bin/hadoop namenode -format'
done
/ProvBase/hadoop/bin/hadoop namenode –format
```

HBase Configuration

The HBase configuration consists in modify a single file inside named name hbase-site.xml that can be found inside the folder conf that is located in the hbase folder. /hbase/conf/

There are several configurations that can be used for HBase depending in the cluster structure and geography. For a simple *ProvBase* cluster, it is recommended to use the following code:

```
<?xml version="1.0"?> <?xml-stylesheet type="text/xsl"
href="configuration.xsl"?>
<configuration>
<property>
   <name>hbase.master</name>
   <value>node1.loca:60000</value>
</property>
<property>
  <name>hbase.rootdir</name>
  <value>hdfs://node1.local:50001/hbase</value>
</property>
<property>
   <name>hbase.cluster.distributed</name>
   <value>true</value>
</property>
</configuration>
```

Installing *ProvBase*

ProvBase comes with a small Installation script that will compile all the necessary files and save them in a folder name bin. This script can be found in the folder SRC where in the previous steps the *ProvBase* code was copied, with the name *install.sh /ProvBase/src*

To execute the script, type:

./install.sh

Starting *ProvBase*

To start *ProvBase*, it is necessary to run a script named startsystem that is located in the folder bin inside the *ProvBase* project.

The script will start Hadoop and HBase. Hadoop will start all the other nodes and HBase will start HBase Database services in the rest of the nodes, according its configuration.

When the start process is complete, execute the Java program client.class. This will initialize a small client when the connection is finished and if everything else is correct, the screen will prompt a message of "Welcome to *ProvBase*" and a menu of options.

BIOGRAPHICAL SKETCH

Jaime Alberto Navarro Yerena was born in Mexico City, Mexico. He attended high school at the Instituto Tecnológico y de Estudios Superiores de Monterrey Campus Hidalgo and graduated in May of 2002.

In August of 2002, Jaime moved to Queretaro, Mexico and started his bachelor's degree in Computer Engineering at the Instituto Tecnológico y de Estudios Superiores de Monterrey Campus Queretaro. In August of 2004, he transferred to the University of Texas-Pan American where he obtained his bachelor's in Computer Science with a minor in Applied Mathematics. In August of 2007, he started his Master's Degree in Computer Science while he was working as a teacher's assistant in the Department of Computer Science at the University of Texas-Pan American.

In January 2008, Jaime postponed his master's degree in order to work full-time as a Programmer Analyst at the Department of Internet Services, University of Texas-Pan American. In 2009, Jaime returned to his studies in order to complete his master's; during this time he began working with Dr. Artem Chebotko on distributed RDF data management technologies.