SPARQL-to-HiveQL Translation

By

Naila Karim

2009-NUST-MS-PhD IT-34

Supervisor

Dr. Khalid Latif

Department of Computing

A thesis submitted in partial fulfillment of the requirements for the degree of Masters of Science in Information Technology (MS IT)

In

School of Electrical Engineering and Computer Science,
National University of Sciences and Technology (NUST),
Islamabad, Pakistan.

(November, 2012)

Approval

It is certified that the contents and form of the thesis entitled "**SPARQL-to-HiveQL Translation**" submitted by **Naila Karim** have been found satisfactory for the requirement of the degree.

Advisor: **Dr. Khalid Latif**
Signature: ____________________

Date: ____________________

Committee Member 1: **Dr. Zahid Anwar**

Signature: ____________________
Date: ____________________

Committee Member 2: **Dr. Sharifullah Khan**

Signature: ____________________
Date: ____________________

Committee Member 3: **Dr. Aimal T. Rextin**

Signature: ____________________
Date: ____________________

Abstract

Resource Description Framework (RDF) is a W3C Recommendation for knowledge representation on semantic web. Growing size of RDF annotated data demands scalable semantic stores. Hadoop based distributed and parallel processing frameworks such as HBase and Hive are increasingly becoming popular for storing voluminous data and for enhancing flexibility to handle complex data. Hive is a Hadoop based data warehousing infrastructure with support for complex analytical processing. Its query interface doesn't support data exploration using SPARQL, a standard query language for RDF. Integration of aforementioned technologies with added support for SPARQL queries may realize a scalable semantic web data store. We have proposed a semantic preserving SPARQL-to-HiveQL translation scheme that adds querying interface to the Hadoop based RDF stores. Major contributions of our research are (i) semantic preserving SPARQL-to-HiveQL query translation algorithm (ii) a storage schema independent querying mechanism that accommodates different storage schemes without impacting translation time. The experimental results show the efficient working of proposed translation algorithm and its support for different types of SPARQL queries.

Certificate of Originality

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any degree or diploma at NUST SEECS or at any other educational institute, except where due acknowledgement has been made in the thesis. Any contribution made to the research by others, with whom I have worked at NUST SEECS or elsewhere, is explicitly acknowledged in the thesis.

I also declare that the intellectual content of this thesis is the product of my own work, except for the assistance from others in the project's design and conception or in style, presentation and linguistics which has been acknowledged.

Author Name: **Naila Karim**
Signature: ________________

Acknowledgment

I am thankful to Almighty Allah for granting me persistence and ability to complete MS thesis.

I wish to thank Dr. Khalid Latif for giving me the opportunity to work under his supervision. I am truly indebted to him for his support during my thesis time. Along with providing me technical help he taught me the importance of independent learning and continuous effort. In equal parts i am thankful to Dr. Zahid Anwar and Mr. Owais A. Malik for arousing my interest in the world of big data processing and getting me started in this domain. Further i wish thanks to Dr. Sharifullah Khan and Dr. Aimal T. Rextin for being my committee members and reviewing this thesis. I am also thankful to my colleague Mudassar for his guidance and contribution in this thesis.

I would also like to send special thanks to all my family members and friend Jaweria. They all were a great source of motivation and mental relaxation for me during all this time.

This work was partially supported by grant from Kyung Hee University under NK-STEP academic collaboration program.

Naila Karim

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction and Background

This chapter gives an introduction of the overall work done and it includes briefing about the technologies used.

1.1 Introduction

Resource Description Framework (RDF) (Manola et al., 2004) is the fundamental building block of Web of Data, also known as Semantic Web (Lee et al., 2001). It provides a standardized framework for data representation and integration. The flexibility offered by RDF has gotten it widely accepted in the Semantic Web community and as a result many storage solutions have emerged (Guo et al., 2005). Persistent RDF stores could be classified as 1) native stores such as Jena TDB (Wilkinson et al., 2003), Sesame Native (Broekstra et al., 2002), AllegroGraph (AllegroGraph, 2012), and OpenLink Virtuoso (Erling, 2001); and 2) non-native stores that work as an overlay on another storage system such as a Relational Database (RDB). In the later case, an RDF store has to translate RDF queries encoded as SPARQL (Prud Hommeaux and Seaborne, 2008) into a query language supported by the underlying storage system such as SQL for RDB. RDF data is growing very rapidly. To some extent, this trend is attributed to Linked Open Data initiative (Bizer et al., 2009a). The DBPedia project (Bizer et al., 2009b), aimed at extracting structured information from infobox section of Wikipedia articles, is a remarkable example of huge RDF data set. Currently it describes 2.6m entities and covers 4.7b pieces of information ranging diverse domains. Freebase (Freebase, 2012) is another open repository of structured data of almost 20m entities. Web Data Commons (WebDataCommons, 2012) is the largest and most up-to-date web corpus, currently available to public. According to the extraction results of February 2012, it contains 1.22b typed entities and 3.29b triples. Read and write operations on such large data sets are increasingly becoming difficult to manage using traditional approaches (Husain et al., 2010). The avalanche of RDF data demands a distributed scale out storage and processing solution that exploits flexible graph structure of RDF data. The big challenge in developing such distributed storage and processing environment is fair data and process distribution. Adding fault tolerance, high availability and load balancing are few of many daunting challenges in building such storage system.

Hadoop is an open source cloud computing framework that is designed on MapReduce programming paradigm and runs on top of low price commodity servers (Apache Hadoop Website, 2012), (Dean and Ghemawat, 2008). Hadoop approach towards distributed data management has been tested and proven to be effective in many domains including information retrieval (Lin and Dyer, 2010), statistical machine translation (Dyer et al., 2008), image processing (Zhang et al., 2010) and stream processing (Kumar et al., 2010). Many enterprises including Yahoo, Amazon, StumbleUpon, LinkedIn and Facebook are using Hadoop for data and processing intensive tasks (Hadoop Wiki Website, 2012). HBase is Hadoop based column store, that has the capability of handling sparse data with row level read and write access (Apache HBase Website, 2012a). These features make it a suitable storage system for RDF data management. HBase has a very limited set of data exploration commands including get, put, scan and delete. Therefore complex operations such as joins need to be implemented as customized MapReduce jobs (Apache HBase Website, 2012b). In contrast SPARQL supports complex queries over RDF data using a rich set of operators and functions. Therefore implementation of all possible SPARQL constructs over HBase as efficient MapReduce jobs remains a challenging task.

Hive is a Hadoop based open source warehousing infrastructure, that supports complex analytics using Hive Query Language (HiveQL) over tons of data stored in Hadoop Distributed File System (HDFS) (Apache Hive Website, 2012). Hive has made the tedious and complex job of writing MapReduce jobs quite easy by automatically generating them for queries written in HiveQL. It automatically handles joins and other commonly used operations in queries. Hive can intelligently handle multiple joins in less number of MapReduce jobs (Thusoo et al., 2010). Enterprises heavily rely on Hive for generating MapReduce jobs for their analytical tasks. At Facebook, 95% of data analytics MapReduce jobs are being generated using Hive. It has increased the productivity of raw MapReduce jobs because these jobs deal with very low level details and therefore are hard to write such custom programs and difficult to maintain and reuse (Lee et al., 2011).

All demands of emerging RDF stores may be fulfilled using Hadoop framework, provided answers to the following question. How to store the graph oriented RDF data on HDFS? How to map RDF data queries into HiveQL? We have proposed a methodology for adding SPARQL query support to an RDF store built on top of HDFS. Our approach shows how Hadoop, HBase and Hive could be used together to design a scale out Semantic Web store with efficient data storage and retrieval capabilities. We opted HBase to solve the scalability issue and to exploit its flexibility of handling semi-structured data and Hive is selected for its data exploration capabilities. Since SPARQL is the standard language to query RDF data, its support for data extraction from HDFS is vital. The use of HiveQL to query RDF data requires translation of SPARQL queries into that of Hive. The manual process of SPARQL-to-HiveQL translation demands complete knowledge of syntactic and semantic rules of both SPARQL and Hive languages. Along with this, it requires thorough understanding of the underlying data storage organization. We automated the aforementioned error prone, complex and manual process of translation. To achieve this goal we have proposed a translation algorithm, and storage schema mapping and translation procedures. The storage schema mapping procedures create a virtual view of the underlying schema. Thus enable the translation algorithm to generate schema specific fast queries while staying independent of the underlying storage. The segregation of the whole process into schema mapping and translation has made the proposed approach totally independent of the data storage layer. Therefore it can easily be used for many configuration of HBase.

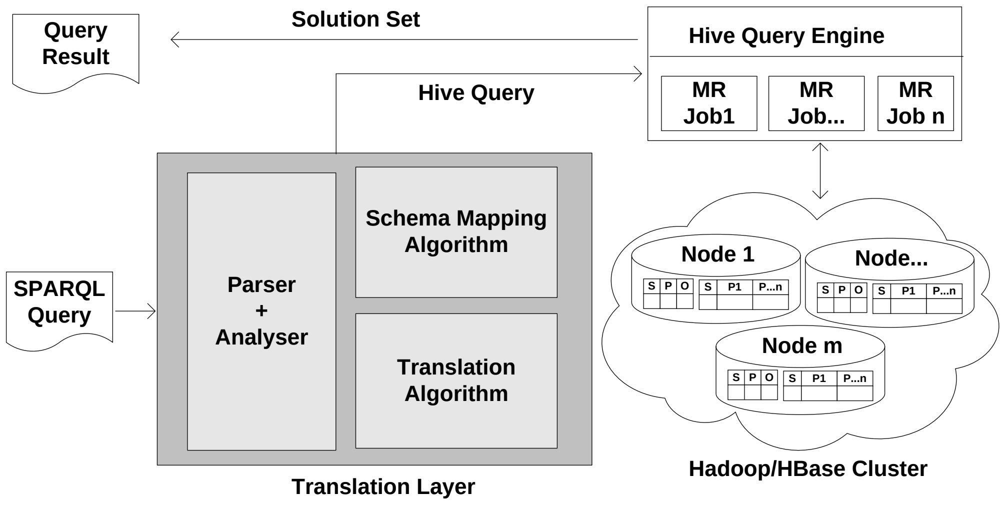

Figure 1.1: System Flow Model

The overall flow of the system is shown in Figure 1.1. The translation process starts after getting an input SPARQL query. The input query is validated using SPARQL parser and a parse tree is generated. The generated parse tree is then passed to the translation algorithm. The translated Hive query is then passed to the Hive query engine. Hive engine automatically translates the Hive query into MapReduce jobs, which are then executed on RDF data stored in HBase to get the solution set.

Our main contributions are,

1. Semantic preserving SPARQL-to-HiveQL query translation algorithm.

2. Storage schema independent querying mechanism that accommodates different storage schemes without impacting translation time.

We conducted extensive experiments to prove the following features of the proposed algorithm: correctness, support of varying SPARQL constructs, efficiency and storage schema independence. We showed the correctness by comparing our generated solution sets with that of Jena. The efficiency, support for various SPARQL constructs and working of the designed algorithm is proved by translating SPARQL queries of varying complexity. To prove storage schema independence, all queries are translated for two different storage schemes. The translation time of algorithm remains in the range of 79ms to 115ms even for lengthy and complicated SPARQL queries irrespective of the underlying storage schema.

1.2 Organization

The rest of the thesis is structured as follows: Chapter 2 discusses the state of the art for our work and the preliminary concepts, around which our methodology is designed. Chapter 3 briefly discusses SPARQL queries. In Chapter 4, we have discussed storage schema mappings in Hadoop. In Chapter 5, we have explained the SPARQL-to-HiveQL translation algorithm along with its working examples. In Chapter 6, we have presented the evaluation of proposed algorithms and Chapter 7 concludes the whole study and presents future work direction.

Chapter 2

Literature Review

This chapter gives a brief overview of the work done in domain of scalable semantic stores and it explains our work in the context of existing work.

2.1 Related Work

With proliferation of Semantic Web applications, tremendous efforts are being made by researchers to build mature RDF stores that could stock up huge amount of RDF annotated data (Sakr and Al-Naymat, 2010). The research work in this domain can be categorized into 1) native RDF stores, 2) RDF storage over RDB, and 3) RDF storage over no-sql data storage systems (such as Hadoop). For category 2 and 3, SPARQL queries - the *lingua franca* for talking to RDF stores (Prud Hommeaux and Seaborne, 2008), should be translated to the language understood by the back end storage system such as SQL for relational databases (Bajda-Pawlikowski, 2008), (Chebotko et al., 2009), (Chebotko et al., 2006), (Elliott et al., 2009a) , (Harris and Shadbolt, 2005), (Lu et al., 2008), (Lv et al., 2010), (Son et al., 2008) and (Zhou and Zheng, 2011). No query translation may be required for the first category of RDF stores, therefore we excluded it from our literature review. Query translation approaches, used in rest of the two categories of RDF stores are discussed subsequently.

2.1.1 SPARQL Over RDB

The research community has proposed and developed mature RDF APIs and stores including Jena (Wilkinson et al., 2003) SDB, Sesame (Broekstra et al., 2002), 3store (Harris and Shadbolt, 2005), (Harris and Gibbins, 2003) and RDFSuite (Theoharis et al., 2005). These stores use mature and vigorous RDB query engine as a back end tool. Since RDF data is based upon graph model, storing it in a RDB has posed many challenges including efficient storage and querying support. Chong (Chong et al., 2005) proposed a SQL based RDF querying approach as a simplistic solution for storing and retrieving RDF data instead of using any other data query language like SPARQL. He introduced an *RDF-Match* table function in SQL to search an intended SPARQL graph pattern. He also added support for RDFS inferencing and rules-based reasoning. This approach strives for reduced query response time at the cost of ignoring advanced SPARQL constructs such as Optional Graph Patterns. Since SPARQL is the *de facto* standard Semantic Web query language, its support in RDF stores is very critical. This in turn requires translating SPARQL queries into equivalent SQL queries for RDB backed RDF stores. Cyganiak (Cyganiak, 2005) has discussed logical equivalence of SPARQL and RDB constructs. He provided grounds for SPARQL to SQL translation by presenting transformation of SPARQL constructs into semantically equivalent relational algebra operators and hence their translation into SQL. Moreover, he outlined the semantic mismatches among different SPARQL and SQL constructs, but the discussion on practically mapping SPARQL data into a RDB considering storage schema design was left untouched. Harris and colleagues, unfolded implementation of translating SPARQL queries into relational algebra for the 3store system (Harris and Shadbolt, 2005). Their work misses out support for nested optional, union and complex value constraints. SPARQL filter constraints to SQL translation is discussed in detail by Lu (Lu et al., 2008). Chebotko and colleagues proposed an algorithm for basic and optional graph pattern translation into SQL for triple-table based RDB storage schema (Chebotko et al., 2006). The assumption that RDF data is stored only using triple-table approach is very limiting and not optimal (Abadi et al., 2009). Recent studies have considered storage schema mapping along with the translation process and have proposed relational functions to accommodate different storage layouts (Chebotko et al., 2009; Elliott et al., 2009b). Chebotko's recent work (Chebotko et al., 2009), notable among other similar efforts, is based on equivalence of relational algebra operations with semantics of SPARQL query operations as outlined in the latest W3C recommendation (Prud Hommeaux and Seaborne, 2008; W3C Website, 2012). They proposed two mapping functions α and β for generating underlying storage schema information, although the results demonstrate efficiency of their approach, but it is evaluated for only triple-table storage schema. The work discussed in (Elliott et al., 2009b) is an extension of the aforementioned approach. It has added support for all SPARQL constructs and generated flat SQL queries instead of nested queries, but it is useful for RDF-LIB style storage schema. Most of the translation algorithms discussed this far generate SQL query considering a fixed storage schema layout. The translation work independent of the underlying storage schema follows the four-step model (Lv et al., 2010; Son et al., 2008). Step one covers SPARQL query pre-processing such as parsing. The translation is performed in second step. SQL post-processing such as query simplification and optimization based on general principles of relational algebra and equivalence rules is performed in third step. The fourth and final step targets query optimization based on the underlying logical and physical data model.

The research work presented in (Lv et al., 2010) and (Son et al., 2008) has suggested that, translation should be kept independent of the underlying storage schema. Instead of tuning a query as per underlying storage scheme their work focuses on generating query specific view of the relational database. The hypothesis is that complexity of the storage design shouldn't influence the translation step as it may require extra work for adapting new storage schemes. Both view based approaches support only a subset of SPARQL queries. Zhou proposed a mechanism for execution of SPARQL queries over heterogeneous databases with the help of uniform SPARQL query interface, though only simple filters, optional and union SPARQL queries are supported (Zhou and Zheng, 2011). A manual tool based approach (Bajda-Pawlikowski, 2008) for SPARQL to SQL translation also exists, though it works only for Triple store, Property store and vertically partitioned storage structure but it supports limited SPARQL constructs.

2.1.2 SPARQL Over Distributed Stores

The research work discussed in previous section covers only moderate sized RDF data repositories. These systems support query processing and reasoning using a single node architecture. Such approaches do not scale well for emerging data intensive semantic enabled applications. Another challenge in using RDB for handling Semantic Web data is the mismatch in the relational model and RDF. RDF follows a graph data model and is very flexible, while relational model requires a strict schema definition. Therefore most suitable solution would be a distributed RDF store with distributed query processing capabilities and flexible schema definition. The research community has already been investigating such distributed solutions. The studies (Husain et al., 2010; Farhan Husain et al., 2009; Franke et al., 2011; Sridhar et al., 2009; Sun and Jin, 2010) have concluded that Hadoop based technologies are most suitable for storing semi-structured and growing RDF data. Hussain and colleagues conducted two different studies targeted at semantic data management in distributed manner (Husain et al., 2010; Farhan Husain et al., 2009). They proposed Hadoop based infrastructure for storage and retrieval of large RDF graphs. They have presented a storage schema for RDF over HDFS by dividing and storing data in different files using two steps. Data is first divided on the basis of predicates called *PS Split* and secondly data is further divided on the basis of explicit *Type* information of objects called *POS Split*. Along with the data storage schema design over Hadoop, they presented an algorithm for determining best execution plan and cost model to execute a SPARQL query as a MapReduce job. Since the data storage scheme presented in (Farhan Husain et al., 2009; Husain et al., 2010) is data dependent thus its performance would degrade if RDF data is skewed, as it would generate unbalanced partitions. The aforementioned scheme works efficiently for basic graph pattern queries, but it lacks support for other SPARQL constructs. MapReduce approach could be used for efficient storage management and retrieval of huge RDF data. Sridhar, for instance, used Pig Latin (Gates et al., 2009), a high level data flow language based on MapReduce (Olston et al., 2008), for processing RDF data analytics (Sridhar et al., 2009). To add support for complex analytical queries they have proposed few extensions. Pig Latin and Hive both are used as automatic MapReduce job generators. A study shows that Hive out performs Pig Latin in terms of efficiency (Jia and Shao, 2009).

Column oriented stores are most suitable for Semantic Web data (Abadi et al., 2009; Weiss et al., 2008). Abadi and colleagues have also proposed that property-table storage schema for RDF data storage reduces the number of joins required to answer queries. The work reported in (Franke et al., 2011) and (Sun and Jin, 2010) suggest that HBase is quite suitable for RDF data storage, since its use resolves the issue of null values and at the same time it exploits the semi-structural properties of semantics web data.

Most of the work reported so far has focused the use of RDB for Semantic Web data, this led the work on SPARQL to SQL translation. The steady increase in amount of RDF annotated data is an evidence that distributed systems are necessary and critical for managing big RDF data-sets. Some researches have suggested the use of cloud based technologies for RDF data handling, but their work mainly focuses the storage schema design, while querying support part hasn't got much attention (Franke et al., 2011; Sun and Jin, 2010). Different techniques for building distributed query engines, using Hadoop (Husain et al., 2010; Farhan Husain et al., 2009) are also suggested in the literature. To the best of our knowledge, there is no effort towards utilizing the existing implementation of MapReduce such as Hive for a scalable and explorable Semantic Web data store.

Chapter 3

SPARQL and Hive

This chapter gives a brief overview of the preliminaries concepts of Hive and SPARQL pertinent to the proposed methodology.

3.1 SPARQL Queries in context of Hive

SPARQL is a W3C recommendation for RDF data (Prud Hommeaux and Seaborne, 2008). In general a SPARQL query has two parts: 1) query type followed by query variables, and 2) WHERE clause followed by graph pattern. According to W3C a query type specifies one of the four possible SPARQL queries including SELECT, ASK, DESCRIBE, and CONSTRUCT. Whereas, a graph pattern specifies query semantics. A graph pattern is one of the following types: Basic Graph pattern (BGP), Basic Graph Pattern with Filter Constraints (FGP), Optional Graph Pattern (OGP), Alternative Graph Pattern (AGP) or UNION Graph Pattern (UGP) and Group Graph Pattern (GGP). A BGP, FGP, and OGP is composed of one or more triple patterns, while AGP/UGP and GGP are made of one or more BGPs, FGPs or OGPs. A triple pattern has three parts subject, predicate and object, called triple parts or query variables. In a SPARQL query a variable is either bound or unbound. A variable is considered bound, if its value is already specified, otherwise it is unbound variable. In contrast to SPARQL, Hive query in general has three parts: 1) query type 2) tables references, and 3) WHERE clause. During translation process a graph pattern in SPARQL is translated to constraints in WHERE clause and join conditions in FROM clause of Hive query. Figure 3.1 shows the partial structure of the SELECT query tree of BNF grammar for both SPARQL and Hive. These trees are generated by our SPARQL and Hive parsers, for the BNF grammar of respective languages using ANTLR (ANTLR Website, 2012). The Figure 3.1

shows the transformation of SPARQL query tree parts into that of Hive.

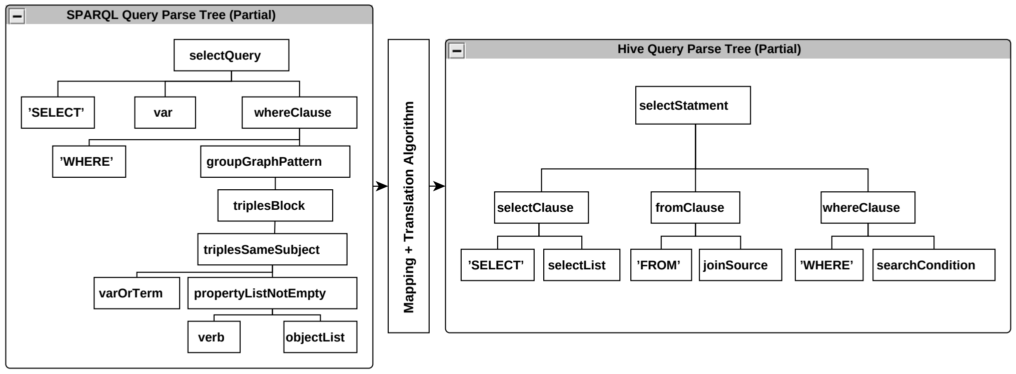

Figure 3.1: An Example of SELECT Query Parse Tree

Graph pattern is most crucial building block of any SPARQL query. The solution set of a query depends on matching the Graph pattern. The Hive equivalent for each type of Graph pattern is discussed below.

- **Basic Graph Pattern** (BGP): Since a BGP may consists of single triple pattern or a set of triple patterns that must match. Hive equivalent query for a single triple BGP is simple, in which all restrictions are specified in the WHERE clause. Multiple triple patterns in a SPARQL query may require specification of restrictions in the WHERE as well as FROM clause of an equivalent Hive query.

- **Basic Graph Pattern with Filter or Value Constraints** (FGP): A BGP with filter further restricts a solution set by specifying more constraints. In Hive, value constraints are specified in WHERE clause.

- **Optional Graph Pattern** (OGP): An OGP defines optional inclusion instead of eliminating solution bindings. In Hive the semantics of OGP could be implemented using LEFT OUTER JOIN.

- **Alternative Graph Pattern** (UGP) or (AGP): It allows the solution set to be obtained by combining the results of different BGPs. In Hive, a UGP/AGP could be implemented by using UNION ALL.

- **Group Graph Pattern** (GGP): A GGP is combination of one or more of the above explained graph patterns. Therefore its translation is done using the above mentioned points.

All operator and functions of SPARQL could be mapped to semantically equivalent operators and functions. After analysing the semantics of SPARQL and Hive operators and functions, we prepared the mapping list shown in Table 3.1:

Table 3.1: Filter Operators and Functions Mapping in Hive

SPARQL Operator/Function	Hive Mapping
+A	A=A+1
-A	A=A-1
A ‖ B	A OR B
A && B	A AND B
A = B	A = B
A != B	A != B
A < B	A < B
A > B	A > B
A ≤ B	A ≤ B
A ≥ B	A ≥ B
A ∗B	A ∗B
A//B	A//B
A + B	A + B
A - B	A - B
Bound(A)	A is NOT NULL
!Bound(A)	A IS NULL
STR(A)	cast (A as STRING)
DATATYPE(A)	cast (A as DATATYPE)
isURI or isIRI	A REGEXP B
REGEX(STRING A, PATTERN B)	A REGEXP B

Chapter 4

Storage Schema

This chapter describes different storage schemes for stocking up RDF data in scalable way. It also discusses our designed storage scheme mapping generation algorithms. The mapping algorithms are integral part of the translation process.

4.1 Storage Schema Mapping in Hadoop

As discussed earlier, column oriented stores are most suitable for efficient storage and retrieval of Semantic Web data (Abadi et al., 2009; Weiss et al., 2008). A group of researchers (Franke et al., 2011; Sun and Jin, 2010) took this hypothesis one step further and proved that HBase could be used as highly scalable, efficient and fault tolerant Semantic Web storage systems. After reviewing the work done for designing a scalable RDF store, we classified it in two broad categories: (i) Static Schemes and (ii) Dynamic Schemes. Since we are using HBase and Hive on top of HDFS, therefore we made the HBase schema and updates visible in Hive as external tables. Furthermore to keep the translation process independent of the data storage structure, an algorithm for generating a storage schema view is essential. Contrary to the storage view generation approaches (Lv et al., 2010; Son et al., 2008), we generate a virtual view. The Algorithm 4.3 is responsible for creating the virtual view. It uses mapping procedures presented as Algorithms 4.4, 4.5, 4.6 and 4.7. The mapping Algorithm 4.3 also uses a helper function isVariable. The function isVariable($triplePart$) takes a triple part and returns true if it is a SPARQL variable and false otherwise. The mapping algorithm receives a data structure $tableName(SubRef, PreRef, ObjRef)$ from the helper procedures and returns it to the translation algorithm. The data structure is made of the following four pieces of information: $tableName$, name of the table in which a triple match for the given triple pattern $triplePattern(s, p, o)$ could be found, where $SubRef$, $PreRef$ and $ObjRef$ are the storage references to the columns for $triplePattern.s$, $triplePattern.p$ and $triplePattern.o$ parts of the triple pattern. In succeeding subsections initially we discussed the storage schema categorization and its accessibility in Hive, and then the storage schema mapping Algorithm 4.3 and its helper procedures refereed as Algorithms 4.4, 4.5, 4.6 and 4.7 are discussed.

4.1.1 Procedures to make HBase table visible in Hive

In this subsection, we have discussed the storage classification and its accessibility in Hive.

Static Schemes

Storage schemes that map RDF data to a fixed schema structure come under this category. The schema layout of such approaches do not evolve even after data changes. There are two known static storage schemes, one is triple-table (Broekstra et al., 2002; Chong et al., 2005; Harris and Gibbins, 2003; Wilkinson et al., 2003) and the other is Hexa store approach (Sun and Jin, 2010; Weiss et al., 2008). The triple-table approach with a single three columnar table to store *subject*, *property*, and *object* parts is the most simplistic and widely used storage scheme for storing RDF data. Many traditional and mature RDF data management systems including Jena (Wilkinson et al., 2003), Sesame (Broekstra et al., 2002), 3store (Harris and Gibbins, 2003) and Oracle (Chong et al., 2005) are using its variants. Structure of a triple-table is depicted through an example in Table 4.1. The other type of static and

Table 4.1: Triple-Table Example

Row Key	ColumnFamily:triple		
unique value	triple:subject	triple:Property	triple:object
1	Ali	type	Student
2	Ali	memberOf	IEEE
3	Ali	year	4th
4	Asher	type	Research Associate
5	Asher	memberOf	IEEE
6	Asher	officeExt	1234

ontology independent storage scheme is introduced in (Weiss et al., 2008) and its implementation using HBase is discussed in study (Sun and Jin, 2010). All RDF data is mapped to six tables *S_PO*, *SO_P*, *P_SO*, *SP_O*, *SO_P* and *PO_S*. The table name shows the storage order of subject (S), predicate (P) and object (O) within a triple. All tables share similar three column structure as shown in Table 4.1, but have different indexing pattern.

The variations in indexes allows efficient searching for different types of triple patterns, though posing a huge overhead on storage because of redundancy.

Algorithm 4.1 GenerateStaticSchema

```
Require: List of HBase tables $t = table_{t1}, table_{t2}, \dots, table_{tn}$
Ensure: HBase tables are accessible through Hive
1: for $table_{t1}$ to $table_{tn} \in t$ do
2:     Probe HBase for $table_t i$ structure
3:     table structure is returned in the form $\langle table_{ti}, (column_{ci1}, \dots, column_{cin}) \rangle$
4:     expose table $table_{ti}$ in Hive as external table $table_{hti}$
5:     for $j = 1$ to all $j = n \in i$th two-tuple do
6:         Add column $column_{cij}$ to Hive table $table_{hti}$
7:     end for
8: end for
```

Making an ontology independent storage scheme accessible in Hive is simple. It requires declaring HBase tables in Hive as external tables. It makes a static HBase storage accessible in Hive. The Algorithm 4.1 takes a list of HBase tables to be manged by Hive as input and probes HBase for getting the table structure information. The table structure information is a two-tuple set member of the form $\langle table_{ti}, (column_{ci1}, \dots, column_{cin}) \rangle$. Where, $1 \leq i \leq n$. In the aforementioned two tuple $table_{ti}$, is the *ith* HBase table with list of columns $(column_{ci1}, \dots, column_{cin})$. The Algorithm 4.1 exposes $table_{ti}$ in Hive as an external table named as $table_{hti}$. In next step it defines the structure of the Hive table $table_{hti}$ by adding all columns $(column_{ci1}, \dots, column_{cin})$ as per table $table_{ti}$ structure. In this manner it makes all HBase tables assessable through Hive.

Dynamic Schemes

A storage structure that evolves with change in data falls under this category. Such schemes are mostly ontology dependent, i.e. for different ontologies the final storage layout might turn out to be completely different but obey similar principles. Property tables (Abadi et al., 2009) as well as the predicate family structure presented in (Franke et al., 2011) are its prominent examples. In property table approach a separate table is created for each unique property. Each table has two columns one for subject and the other for storing object part. The general structure of a property table is shown in Tables 4.2, 4.3, 4.4 and 4.5.

Table 4.2: Property Table:Type

subject	object
Ali	Student
Asher	Research Associate

Table 4.3: Property Table:memberOf

subject	object
Ali	IEEE
Asher	IEEE

Table 4.4: Property Table:year

subject	object
Ali	4th

Table 4.5: Property Table:officeExt

subject	object
Asher	1234

The other variant of ontology dependent scheme is predicate-table. In this approach a table stores all property values of a subject as key-value pairs in one row. Hence row count of the table is equal to the number of distinct subjects in RDF data. The structure of predicate-table approach is demonstrated through an example in Tables 4.6 and 4.7.

Table 4.6: Predicate-Table: SPO

Row Key	ColumnFamily:predicate		
subject	predicate:p1	predicate:p2	predicate:pn
Ali	type	year	memberOf
	Student	4th	IEEE
Asher	type	officeExt	memberOf
	Research Associate	1234	IEEE

Table 4.7: Predicate-Table:OPS

Row Key	ColumnFamily:predicate		
object	predicate:p1	predicate:p2	predicate:pn
Student		type	
		Ali	
4th		Year	
		Ali	
IEEE		memberOf	
		Ali	
		Asher	
Research associate		type	
		Asher	
1234		officeExt	
		Asher	

In predicate-table approach data is replicated in two tables to optimize searches on both subjects and objects. The replicated table only takes objects as row keys contrary to subjects as used in the primary predicate-table. An example is listed in Table 4.7. Algorithm 4.2 is used for making an ontology dependent storage structure visible in Hive. The algorithm requires the list of HBase tables to be manged by Hive and the type of the storage structure as arguments. It checks if the schema type is predicate-table, then it creates two tables SPO and OPS. In SPO a column named *subject* is added

Algorithm 4.2 GenerateDynamicSchema

```
Require: List of HBase tables $t = table_t1, table_t2, \ldots, table_tn$, Storage Schema Type $\tau$
Ensure: HBase tables are accessible through Hive
 1: if $\tau \mapsto$ predicate-table then
 2:   Expose table $SPO$    $\triangleright$ $SPO$ is a table with subjects as row key, predicates as columns, objects as cell values
 3:   Add column subject
 4:   Add column predicate $\langle propertyAsKey, objectASValue \rangle$    $\triangleright$ It will automatically add all columns
 5: else
 6:   if $\tau \mapsto$ predicate-table then
 7:     Expose table $OPS$    $\triangleright$ $OPS$ is a table with object values as row keys, predicates as columns, subjects as cells
 8:     Add column object
 9:     Add column predicate $\langle propertyAsKey, subjectAsValue \rangle$    $\triangleright$ It will automatically add all columns
10:   end if
11: else
12:   if $\tau \mapsto$ Property Table then
13:     for $table_t1$ to $table_tn \in t$ do
14:       Expose table $table_{hti}$ in Hive with name $table_{ti}$
15:       add column subject
16:       add column object
17:     end for
18:   end if
19: end if
```

to store all distinct subjects, while all other predicate and object parts are added as a map $\langle propertyAsKey, objectAsValue\rangle$. In map predicate names are stored as keys and objects are stored as values. In OPS *object* is the first column, while all other columns are added as a map$\langle propertyAsKey, subjectAsValue\rangle$. In map predicate names are stored as keys and subjects are stored as values. In this manner the tables SPO and OPS are exposed as Hive external tables. In second case if the schema type is Property Table, then the Algorithm 4.2 simply exports all the tables as a two columnar matrix, where one column is named as *subject* and the other is *object*.

4.1.2 Storage Schema Mapping Algorithm

In this subsection we will explain the algorithm responsible for creating virtual storage schema view and its helper procedures. The Algorithm 4.3 takes as arguments, the triple pattern $triplePattern(s, p, o)$ and the target storage schema type *schemaType* and depending upon the schema type it calls an appropriate helper mapping procedure. The called procedure creates a virtual view of the *schemaType* for the triple pattern $triplePattern(s, p, o)$, and returns the mapping information as a data structure $tableName(SubRef,\ PreRef, ObjRef)$ to the calling Algorithm 4.3.

Algorithm 4.3 GenerateSchemaMapping

```
Require: $schemaType$, $triplePattern(s, p, o)$
Ensure: Schema mappings are generated and aliases are assigned
 1: if $schemaType = TripleTable$ then
 2:   return $TripleTableSchemaMapping()$
 3: else
 4:   if $schemaType = HexaStore$ || $schemaType = PropertyTable$ && is-
      Variable(p)=true then
 5:     return $HexaStoreSchemaMapping(triplePattern(s, p, o))$
 6:   end if
 7: else
 8:   if $schemaType = PropertyTable$ && isVariable(p)=false then
 9:     return $PropertyTableSchemaMapping(triplePattern(s, p, o))$
10:   end if
11: else
12:   if $schemaType = PredicateTable$ then
13:     return $PredicateTableSchemaMapping(triplePattern(s, p, o))$
14:   end if
15: end if
```

Now we will discuss the helper procedures used by Algorithm 4.3 in schema mapping. The helper procedures named as Algorithms 4.4 and 4.5 are used for mapping a given triple pattern to a storage structure for a static storage scheme.

The algorithm 4.4 creates mapping, if the data is in triple-table format. It maps a given triple pattern to a table $TripleTable$. The subject part of triple pattern to column *subject*, similarly predicate part is mapped to column *predicate* and object part

Algorithm 4.4 TripleTableSchemaMapping

```
Ensure: Schema mapping is generated.
1: $tableName \leftarrow 'TripleTable'$
2: $\acute{s} \leftarrow 'subject'$
3: $\acute{p} \leftarrow 'predicate'$
4: $\acute{o} \leftarrow 'object'$
5: return $tableName\ (\acute{s}, \acute{p}, \acute{o})$
```

to column *object*. After generating mapping, it returns the data structure: *tableName* (*SubRef*,*PreRef*,*ObjRef*). The data structure is described earlier.

Algorithm 4.5 HexaStoreSchemaMapping

```
Require: $triplePattern(s, p, o)$
Ensure: For a given a $triplePattern$ schema mapping is generated.
1: if $isVariable(s) = false$ then
2:     $given =' S'$
3: else
4:     $var =' S'$
5: end if
6: if $isVariable(p) = false$ then
7:     $given = concat(given,' P')$
8: else
9:     $var = concat(var,' P')$
10: end if
11: if $isVariable(o) = false$ then
12:     $given = concat(given,' O')$
13: else
14:     $var = concat(var,' O')$
15: end if
16: $tableName = concat(given,' \_', var)$
17: $\acute{s} \leftarrow 'subject'$
18: $\acute{p} \leftarrow 'predicate'$
19: $\acute{o} \leftarrow 'object'$
20: return $tableName\ (\acute{s}, \acute{p}, \acute{o})$
```

The algorithm 4.5 is responsible for generating a virtual view of the physical storage, when data is in a Hexa store. It directs data retrieval to different tables, depending upon the return value of helper function isVariable. If isVaraible returns true for one of the subject s, predicate p or object o part of the *triplePattern*, then *tableName* is mapped to one of the tables *S_PO*, *P_SO*, *SO_P* respectively. Similarly if isVaraible() returns true for for any two of the subject s, predicate p or object o parts of triple pattern, then *tableName* is mapped to one of the tables *SO_P*, *SP_O*, *PO_S*. All tables in Hexa store share a three columnar structure. Therefore subject, predicate and object parts are always mapped to *subject*, *predicate* and *object* table columns respectively. The algorithm 4.5 returns a data structure *tableName*(*SubRef*,*PreRef*,*ObjRef*) to the calling procedure.

Algorithm 4.6 PropertyTableSchemaMapping

```
Require: $triplePattern(s, p, o)$
Ensure: Schema mappings are generated and aliases are assigned
1: $tableName \leftarrow p$
2: $\acute{s} \leftarrow 'subject'$
3: $\acute{p} \leftarrow tableName$
4: $\acute{o} \leftarrow 'object'$
5: return $tableName\ (\acute{s}, \acute{p}, \acute{o})$
```

The algorithms 4.6 and 4.7 generate a virtual storage view for dynamic schemes. The algorithm 4.6 is used to generate virtual view for property-table based store. Since the property table storage design is based on the fact, real world SPARQL queries have known predicate values. Therefore this approach is used to efficiently answer SPARQL queries with known predicate values. It directs the query for a given *triplePattern* to the target table *triplePattern.p*, referred as *tableName*. It directs the subject part to the *subject* column and the object part to the *object* column.

The Algorithm 4.7 generates, a storage mapping for the input triple pattern: *triplePattern* (s, p, o), when data is in a Predicate store. It uses the helper function isVarable to determine the mapping for table name. It directs the query to the table *OPS*, if the function isVariable returns true for subject part of the triple pattern *triplePattern.s* and false for the object part *triplePattern.o*. Furthermore using the function isVariable, it generates mapping for all three parts s, p and o of the *triplePattern*. In the table *OPS* a column named *object* stores the object part of triple pattern, therefore it is mapped as *objet* = *triplePattern.o*. The predicates and subject parts are stored as a map named predicate$\langle propertyAsKey, subjectAsValue\rangle$. Therefore if function isVariable is false for predicate part, then predicate is mapped to *predicate*[*p*]. The subject part is mapped to the value of the key as follows *predicate*[*p*] = *triplePattern.s*. In similar manner storage mappings are generated for a given triple pattern, when isVariable returns false for subject part of the triple pattern. In this case *tableName* is mapped to *SPO* and the column mappings for the subject, predicate and object part of triple pattern are generated as explained earlier for table *OPS*.

Algorithm 4.7 PredicateTableSchemaMapping

```
Require: $triplePattern(s, p, o)$
Ensure: Schema mappings are generated and aliases are assigned
 1: if isVariable(s)=true && isVariable(o)=false then
 2:     $tableName \leftarrow 'OPS'$
 3:     $\acute{o} \leftarrow 'object'$
 4:     if isVariable(p)=false then
 5:         $\acute{p} \leftarrow predicate[p]$        ▷ predicate[p] refers the key of
            map predicate$\langle propertyAsKey, subjectAsValue\rangle$
 6:     else
 7:         $\acute{p} \leftarrow *$
 8:     end if
 9:     $\acute{s} \leftarrow \acute{p}$
10: else
11:     $tableName \leftarrow 'SPO'$
12:     $\acute{s} \leftarrow 'subject'$
13:     if isVariable(p)=false then
14:         $\acute{p} \leftarrow predicate[p]$
15:     else
16:         $\acute{p} \leftarrow *$
17:     end if
18:     $\acute{o} \leftarrow \acute{p}$
19: end if
20: return $tableName\ (\acute{s}, \acute{p}, \acute{o})$
```

Chapter 5

SPARQL-to-HiveQL Translation

This chapter discusses the overall translation process and all its technical peculiarities. Initially parsing details are explored and then all Translation algorithm are explained with examples.

5.1 SPARQL-to-HiveQL Translation

In this section, we will discuss translation of SPARQL queries into semantically equivalent Hive queries using the proposed translation algorithm. First step in translation is validation and identification of query parts. Therefore a query scan using complete grammar rules of input language is crucial. We designed a SPRAQL parser to parse the query for validation. Parser also builds a parse tree, used in query parts identification. This approach saves the time and effort of manual query scan. After query scan, the real process of translation starts.

Our translation algorithm generates a flat SPARQL query for each BGP, this feature results in simple and efficient Hive queries without requiring further processing and simplification. To make the translation process more flexible and adaptable in accordance with W3C SPARQL recommendations, we have designed it around SPARQL graph patterns. The translation process starts in Algorithm 5.1. An input SPARQL query and target schema type are passed as inputs to the Algorithm 5.1. It then, using the helper translation procedures refereed as Algorithm 5.2 and 5.3, and helper mapping procedure refereed as Algorithm 4.3, decodes the passed query into Hive language. An overview of the overall translation process is as follows. The Algorithm 5.1 create a list of unbound SPARQL variables, referred as *required* and another list of bound variables, refereed as *given*. The lists *required* and *given* are created for each BGP using helper translation procedure, Algorithm 5.2. Each list is assigned a sequential graph number using a helper translation function *groupAlias*(). Lists related to one BGP are used to form a flat query, so if there is only one BGP, made of dozens of triple patterns, a flat Hive query is generated. In case, when a group graph pattern is made of more than one nested graph patterns, then initially flat queries are generated for each graph pattern. In next step depending upon the connecting operator or keyword between the graph patterns, flat queries are joined to form a nested Hive query while preserving query semantics. The process of joining together any number of flat Hive queries to generate one query, involves two steps. In step one the translation of the connecting graph pattern operator is performed. In step two, it connects the flat queries using the translated BGP connecting operator, and adds a new SELECT clause. The SELECT clause of nested sub queries is generated by taking union of their required lists. Finally SELECT clause is refined by taking an intersection of the variables in SELECT clause of the grouped Hive query, with that of the *selectClause* list generated by helper translation function $genFormat(sparqlQuery)$. The translation Algorithm 5.1 is also responsible for limiting the level of nesting in the target Hive query. Nesting is limited to the number of participating graph patterns in an input SPARQL query rather than the number of triple patterns.

The in-depth working of the Algorithm 5.1 is as follows. The semantics of a SPARQL is hidden in its WHere clause. Therefore to ensure correct translation, we format it to a form processable by the Algorithm 5.1. In general the WHERE clause of a SPARQL query is a group made of one or more BGPs (BGP_i) connected together through different operators(op_i), where $1 \leq i \leq n$. The BGP connecting operator *op* is a member of the set gOp = {$OPTIONAL$, $UNION$, $NoOperator$}. To make input query processable, it is passed to the helper translation function $genFormat(sparqlQuery)$, which returns the *wherePart* in a format like this: $\{BGP_1, op_1, BGP_2, \ldots, op_n, BGP_n\}$ and, a list of SPARQL query Select clause variables refereed as : *selectPart*. Each operator op_i in *wherePart* is from the set *gOp*. To make the working of function $genFormat(Q_i)$ more clearer, lets process a query using it. The processing of the *whereClause* for a group graph pattern query say $Q_i = \{BGP_1\ OPTIONAL\ \{BGP_2\ OPTIONAL\ \{BGP_3\}\}\}$. The function $genFormat(Q_i)$ process Q_i and returns $wherePart = \{BGP_1, OPTIONAL_1, BGP_2, OPTIONAL_2, BGP_3\}$. After having input query in processable format, it is passed to the Algorithm 5.1. It calls the helper translation procedure, Algorithm 5.2 for each BGP_i in *sparqlQuery* and translate it into a flat Hive query h_{fi}. In next step, it starts connecting the generated flat Hive queries in descending order. Initially generated queries h_{fn} and $h_{f(n-1)}$ are connected using OP_{n-1} to form a nested hive query h_{n-2}. In next step h_{n-2} is connected with $h_{f(n-2)}$ using op_{n-2} to produce h_{n-3}. The process of connecting flat queries is continued until a query h is formed by connecting h_1 with h_{f1} using op_1. The BGPs connecting operator op_i has three possible values in case 1, $op_i = OPTIONAL$ is translated to ($⟕$), and in case 2, $op_i = UNION$ is decoded to $UNION\ ALL$ and in last case 3, $op_i = NoOperator$ is replaced by ($\bowtie$) in the decoded Hive query. For operators in case 1 and case 2 the ($\bowtie$) or ($⟕$), joining condition among two BGPss in a query h is generated by calling the helper translation procedure, Algorithm 5.3. The detailed working of Algorithm 5.3 is given at the end of this section. In case 3 h is generated by connecting flat queries with Hive operator $UNION\ ALL$.

Now we will explain the working of the helper translation procedure, Algorithm 5.2. It is used for resolving the semantics of each BGP. It takes a BGP as input and process it. After processing, it returns two lists named: *required* and *given*.

The Algorithm 5.2 resolves a set of triple patterns(tp_i) and filter patterns(f_i) within a BGP(BGP), where $\geq i \leq$. At first place, it takes $BGP = \{tp_1, f_1, tp_2, f_2, \ldots, tp_n, f_n\}$ as input and using the helper translation function $reOrder()$, reorders it. The helper translation function $reOrder()$ arranges the triple patterns, and filter patterns within a BGP by moving all filter constraints to end of the BGP as follows: $BGP = \{tp_1, f_1, tp_2, \ldots, tp_n, f_2, \ldots, f_n\}$. All $tp_i \in BGP$ and all $f_i \in BGP$ are processed in an iterative manner. To process each $tp_i \in BGP$, it uses the helper mapping procedure, Algorithm 4.3, that generates storage mappings for tp_i. The Algorithm 5.2 then using the mapping in-

Algorithm 5.1 GenerateHiveQuery

```
Require: $sparqlQuery$, $schemaType$
Ensure: $h$
 1: Call $genFormat(sparqlQuery)$
 2: for $i = 1$ to $i = n \in wherePart$ do
 3:    Call $BasicGraphPattern(BGP_i)$
       $required_{fi} \leftarrow required$, $given_{fi} \leftarrow given$
 4:    Add all members of $required_j$ to $project_{fi}$
 5:    for $j = 1$ to $j = n \in project_{fi}$
       $selectClause_{fi} = concat\_ws((if(i > 1),',','SELECT'), project_{ij})$
 6:    $jCond_{fi} \leftarrow genJoinPairs(required_{fi})$
 7:    for $k = 1$ to $k = n \in jCond$
       Add an equality based Join to the $fromClause_{fi}$
 8:    for $k = 1$ to $k = n \in given$
       $whereClause_{fi} = concat\_ws((if(i > 1),',','WHERE'), given_{ik})$
 9:    $h_{fi} = concat\_ws(selectClause_{fi}, fromClause_{fi}, whereClause_{fi})$
10: end for
11: if $i > 1$ then
12:    for $j = i - 1$ to $j = 1$ do
13:       for $k = i - 1$ to $k = j$
          $project_j = project_{f(k+1)} \cup project_{fk}$
14:       for $l = 1$ to $l = n \in project_j$
          $selectClause_{j-1}$=$concat\_ws((if(l > 1),',','SELECT'), project_{jl})$
15:       if $op_j = OPTIONAL$ then
16:          $op$ =$LEFT\ OUTER\ JOIN$
17:       else
18:          if $op_j = UNION$ then
19:             $op$ =$UNION\ ALL$
20:          end if
21:       else
22:          if $op_j = noOperator$ then
23:             $op$ =$JOIN$
24:          end if
25:       end if
26:       $fromClause_{j-1}$=
          $concat\_ws('FROM(', (if(j = i - 1), h_{f(j+1)}, h_j), op, h_{fj},')')$
27:       if $op \neq UNION\ ALL$ then
28:          $jCond_{fi} \leftarrow genJoinPair(project_j)$
29:          for $k = 1$ to $k = n \in jCond$
             Add an equality based Join to the $fromClause_j$
30:       end if
31:       $h_{j-1} = concat\_ws(selectClause_{j-1}, fromClause_{j-1})$
32:    end for
33:    return $h$
34: else
35:    return $h_{fi}$
```

Algorithm 5.2 BasicGraphPattern

```
Require: $BGP$=$\{tp_1, f_1, tp_2, f_2, \ldots, \ldots, tp_n, f_n\}$
Ensure: Two lists $required$ and $given$
 1: Initialize lists $required$ , $given$=$\emptyset$
 2: reOrder($BGP$) $\leftarrow \{tp_1, tp_2, \ldots, tp_n, f_1, f_2, \ldots, f_n\}$
 3: for $tp_1$ to $tp_n \in BGP$ do
 4:     $tableName(SubRef, PreRef, ObjRef) \leftarrow GenerateSchemaMapping(schemaType, tp_i)$
 5:     genAlias($tp_i$)        ▷ genAlais is responsible for assigning aliases
 6:     if $isVariable(tp_i.subject) = true$ then
 7:         $required$= $required \cup TableAlias.SubRef$ $as$ $var\_subject$
 8:     else
 9:         $given$= $given \cup TableAlias.SubRef = subject$
10:     end if
11:     if $PredStorageRef != TableName$ then
12:         if $isVariable(tp_i.predicate) = true$ then
13:             $required$= $required \cup TableAlias.Ref$ $as$ $var\_predicate$
14:         else
15:             if $isMap(PreRef) = true$ then
16:                 $given$= $given \cup concat\_ws(TableAlias.PreRef,' ISNOTNULL')$
17:             end if
18:         else
19:             if $isMap(PreRef) = false$ then
20:                 $given$= $given \cup TableAlias.PreRef = predicate$
21:             end if
22:         end if
23:     end if
24:     if $tp_i.object \neq tp_i.subject$ then
25:         if $isVariable(tp_i.object) = true$ then
26:             $required = required \cup TableAlias.ObjRef$ $as$ $var\_object$
27:         else
28:             $given$= $given \cup TableAlias.ObjRef = object$
29:         end if
30:     else
31:         $given$= $given \cup TableAlias.SubRef = TableAlias.ObjRef$
32:     end if
33: end for
34: for $f1_1$ to $f_n \in BGP$ do
35:     $hiveCond \leftarrow operatorMapping(sparqlCond)$
36:     $given \cup TableAlias.ColRef(hiveCond))$        ▷ $TableAlias.ColRef$ is
        obtained by looking operand in $required$ list
37: end for
38: return $required$, $given$
```

formation $tableName(SubRef, PreRef, ObjRef)$ and some helper functions $isVariable$ and $isMap()$ creates lists. It creates lists by adding the subject $triplePattern.s$, predicate $triplePattern.p$ and object $triplePattern.p$ parts of tp_i along with the storage information to either *required* or *given*. After processing all $tp_i \in BGP$, it starts to processing the filter constraints $f_i \in BGP$. For this purpose it makes use of the helper function $operatorMapping()$. The function $operatorMapping()$ basically searches for best match of SPARQL filter operator listed in the table 3.1, implemented as a hashmap $\langle SPARQLoperator(key), HiveOperator(value)\rangle$. After translating each filter constraint and getting its operands storage mapping information, the translated filter constraint is added to list *given*.

Finally working of the helper translation procedure refereed as Algorithm 5.3 is discussed. Its purpose is to generate the list of joining conditions for a graph pattern. It takes as input, a list *required* of all unbound variables in a group pattern and generates another list $jCond$. List $jCond$ contains a pair of join operands at its each index position.

Algorithm 5.3 genJoinPairs

```
Require: $required$
Ensure: $jCond$
 1: for $i = 1$ to $i = |required| - 1$ do
 2:     for $j = i + 1$ to $j = |required| - 1$ do
 3:         $var1 \leftarrow varPart(required[i])$
 4:         $var2 \leftarrow varPart(required[j])$
 5:         if $var1 = var2$ then
 6:             $jCond = jCond \cup concat(var1, ',', var2)$
 7:             $break$
 8:         end if
 9:     end for
10: end for
```

The step wise working of the proposed translation algorithm is illustrated with the help of following example.

Example 1: SPARQL Basic Graph Pattern Query

```
SELECT ?item
Where { ?item rdf:type ?type .                    #(tp1)
        ?item mods:subject ?object                #(tp2)
      }
```

In Example 1 translation process of a BGP query is explained in three steps. The input query in Example 1 is translated for four different types of storage schemes. The *Step* 1 shows, storage schema mappings generated for Triple store, Hexa store, Property store and Predicate store using Algorithm 4.3. In *Step* 2 shows the lists *given* and *required* generated using Algorithm 5.2. The *Step* 3 shows the final output Hive queries produced by the Algorithm 5.1. The brief explanation of each steps is given below.

The storage mappings generated by the Algorithm 4.3 using its helper mapping procedures refereed as, Algorithms 4.4, 4.5, 4.6 and 4.7 are shown in *Step* 1. In *Step* 1

Step 1: Generating Schema Mappings

```
Triple store: TripleTable(subject, predicate, object)
          (tp1)
TripleTable(subject, predicate, object)
          (tp2)
Hexa store: P_SO(subject, predicate, object)
          (tp1)
P_SO(subject, predicate, object)
          (tp2)
Property store: rdf:type(subject,rdf:type,object)
          (tp1)
mods:subject(subject,rdf:type,object)
          (tp2)
Predicate store: SPO(subject,predicate['rdf:type'],predicate
   ['rdf:type'])  (tp1)
SPO(subject,predicate['mods:subject'],predicate['mods:subject
   ']) (tp2)
```

Step 2: Generating the Bound and Unbound Variables Lists

```
Triple store: required[t1.subject as var_item, t1.object as
   var_type, t2.subject
as var_item, t2.object as var_object] given[t1.predicate='rdf
   :type', t2.predicate='mods:subject']
Hexa store: required[t1.subject as var_item, t1.object as
   var_type, t2.subject
as var_item, t2.object as var_object] given[]
Property store: required[t1.subject as var_item, t1.object as
    var_type,
t2.subject as var_item, t2.object as var_object] given[t1.
   predicate='rdf:type', t2.predicate='mods:subject']
Predicate store: required[t1.subject as var_item, t1.
   predicate['rdf:type'] as
var_type, t2.subject as var_item, t2.predicate['mods:subject
   '] as var_object] given[t1.predicate['rdf:type'] IS NOT
   NULL, t2.predicate['mods:subject'] IS NOT NULL]
```

the shown storage mapping for *Triple store* is generated using Algorithm 4.4. The Algorithm 4.4 directs $tp1$ to the table *TripleTable*. It also specifies that, $tp1.s = var\_item$ be mapped to the *subject* column of *TripleTable*. Similarly $tp1.p = rdf : type$ and $tp1.o = var\_type$ be respectively mapped to columns *predicate* and *object* of *TripleTable*. In same way, it generates the storage mapping for $tp2$. The triple pattern $tp2$ is also directed to table *TripleTable*, where its parts $tp2.s = var\_item$, $tp2.p = mods : subject$ and $tp2.o = var\_object$ are mapped to *subject*, *predicate* and *object* columns respectively.

The storage mapping for *Hexa store* is generated using the Algorithm 4.5. The Al-

Step 3: Translated Basic Graph Pattern Hive Query

```
Triple store: SELECT t1.subject as var_item
  FROM TripleTable t1 JOIN TripleTable t2 ON(t1.subject=t2.
     subject)
  WHERE t1.predicate="rdf:type" AND t2.predicate=mods:subject
     ";
Hexa store: SELECT t1.subject as var_item
  FROM P_SO t1 JOIN P_SO t2 ON(t1.subject=t2.subject)
  WHERE t1.predicate="rdf:type" AND t2.predicate=mods:subject
     ";
Property store: SELECT t1.subject as var_item
  FROM rdf:type t1 JOIN mods:Text t2 ON(t1.subject=t2.subject
     );
Predicate store: SELECT t1.subject as var_item
  FROM PredicateTable t1 JOIN PredicateTable t2 ON(t1.subject
     =t2.subject)
  WHERE t1.predicate["rdf:type"] IS NOT NULL
  AND t2.predicate["mods3:subject"] IS NOT NULL  ;
```

gorithm 4.5 shows, table name binding may change. Since only predicate is bound in both triple patterns $tp1$ and $tp2$, therefore they are directed to the table P_SO. All tables share the three columnar structure, therefore $tp1.s = var\_item$ be mapped to the column *subject* of table P_SO. Similarly $tp1.p = rdf : type$ and $tp1.o = var\_type$ be respectively mapped to columns *predicate* and *object* of P_SO. In same way, it generates the storage mapping for $tp2$. The triple pattern $tp2$ is also directed to table P_SO, where its parts $tp2.s = var\_item$, $tp2.p = mods : subject$ and $tp2.o = var\_object$ are mapped to *subject*, *predicate* and *object* columns respectively.

Storage mapping for *Property store* is generated using the Algorithm 4.6. Since predicates are bound in both triple patterns $tp1$ and $tp2$, therefore they are mapped to tables $rdf : type$ and $mods : subject$ respectively. Since each table in *Property store* consists of two column, therefore $tp1.s = var\_item$ and $tp1.o = var\_type$ are mapped to the *subject* and *object* column of $rdf : type$ respectively. In similar manner storage mapping for $tp2$ are generated. Its parts $tp2.s = var\_item$ and $tp2.o = var\_object$ are mapped to *subject* and *object* columns of table $mods : subject$ respectively.

Finally for the *Predicate store*, the Algorithm 4.7 generates storage mappings. For the given query both triple patterns $tp1$ and $tp2$ are mapped to the table SPO, as subject and object are unbound in both $tp1$ and $tp2$. In table SPO, *subject* is row key and $predicate < predicate, object >$ is a map, in which predicate is stored as key and object as value. Therefore for $tp1$, the subject part var_item is mapped to the column *subject*, while predicate part is mapped to $predicate['rdf : type']$. Since object part is stored as value for key predicate in the map *predicate*, therefore it is refereed through the map name *predicate* and its key $'rdf : type'$ like this: $predicate['rdf : type']$. Similarly for $tp2$, the subject part var_item is mapped to the column *subject*, while predicate part is mapped to $predicate['mods : subject']$. Since object part is stored as value for key predicate in the map *predicate*, therefore it is refereed as given $predicate['mods : subject']$.

The *Step 2* shows lists of the SPARQL query bound and unbound variables along with their table and column name aliases. In *Example 1* *?item* and *?type* in *tp*1 and *?item* and *?object* in *tp*2 are unbound values, therefore are added to the *required* list, while *rdf* : *type* in *tp*1 and *mods* : *subject* in *tp*2 are bound, so they are added to the *given* list. Similarly for *Hexa store* both lists *required* and *given* are generated in exactly same manner with same members. The *Property store* also has the same *required* list, but *given* is empty here. This difference is due to variation of table structure in *Property store*. Since predicate names are mapped to tables, and are not added to the *given* or *required* list as shown in Algorithm 5.2. The *required* and *given* lists generated for *Predicate store* contains the same members with difference in references of table and column names.

The *Step 3* shows decoded Hive queries generated by algorithm 5.1. The queries are generated using the helper translation procedures and helper mapping procedures. The Algorithm 5.2 is used to generate the lists *required* and *given* in similar way as explained earlier in *Step 2*. The Algorithm4.3 is used to generate the schema mapping as explained in *Step 2* above. In next step the *required* list is passed to the helper translation procedure refereed as Algorithm 5.3, that generates the list of shared variables among the triple patterns. For *Example* 1 query the list is: $jCond[t1.subject, t2.subject]$. Therefore the triples are joined on the subject column. Since the query in *Example* 1 is BGP query, therefor flat Hive query for different storage schemes is shown in *Step: 3 Translated Basic Graph Pattern Hive Query*.

5.1.1 Translation Example of Basic Graph Pattern with Filter Constraint

To understand translation of this genre of queries, lets consider the *Example* 1 query with an added Filter constraint: FILTER($?type! = mods : StillImage$). The translation proceeds in the same fashion as it went for *Example* 1. The only difference is an added element to the *given* list of all four types of storage schemes. The added element in the *given* list for *Triple store, Hexa store, Property store* is $t1.object <> "mods : StillImage"$, while for *Predicate store* it is $t1.predicate["rdf : type"] <> "mods : StillImage"$. The filter constraint is translated using the Algorithm 5.2. The translated *Example* 1 query with an added Filter constraint: FILTER($?type! = mods : StillImage$) is almost same with a little difference in its WHERE clause. The Queries for the first three schema type contains one more bounding condition *And* $t1.object <> "mods : StillImage"$ appended at the end of the WHERE clause. The Hive query for *Predicate store* has a condition *AND* $t1.predicate["rdf : type"] <> "mods : StillImage"$ appended at the end of its Where clause.

5.1.2 Translation Example of Optional Group Graph Pattern

In this subsection translation of the OGP query is explained. The sample *Example* 3 OGP query, in the proposed representation format is: $wherePart = \{BGP_1, OPTIONAL1, BGP_2\}$. Hive queries h_{f1} and h_{f2} are generated for the groups BGP_1 and BGP_2 in similar manner as explained in *Example* 1. The flat queries are connected through operator *OPTIONAL*. As explained earlier, it is translated to Hive *LEFT OUTER JOIN* ($⟕$)). Therefore h_{f1} and h_{f2} are connected using an operator ($⟕$). The join condition is generated using the Algorithm 5.3. For the considered query the join is on the shared variable $g1.var\_item = g2.var\_item$. Finally the refined Select clause is added to project only the

Example 2: SPARQL Optional Group Graph Pattern Query

```
SELECT ?recordID  ?item
Where { ?recordID mods:records ?item .
        ?item rdf:type mods:Text .
        OPTIONAL { ?item mods:otherVersion ?version
           .
                ?item mods:isReferencedBy ?reference
                    } }
```

variables of list *project*. To conserve space and avoid repetition, the completely translated Hive query for only *Triple store* and *Predicate Store* is shown in *listing:Translated Optional Group Graph Pattern Hive Query.*

5.1.3 Translation Example of Alternative Group Graph Pattern

The subsection describes translation process of the AGP/UGP queries. Since the solution set of an AGP/uGP query is the union of all solution sets produced for each graph pattern. Therefore to preserve query semantics, initially flat queries are generated for each group graph pattern. The generated flat Hive queries are then combined using Hive *UNION ALL*. The translation process of a union query for *Triple store* and *Predicate store*, is explained using a sample *Example* 3 query. The Algorithm 5.1 using its helper translation and mapping procedures translates the query. The mentioned query in generalized representation format is: $wherePart = \{BGP_1, UNION, BGP_2\}$. The three step process, as explained earlier for the *Example* 3 query is performed for both BGP_1 and BGP_2 to generate the flat queries h_{f1} and h_{f2}. Since the connecting operator here is *UNION*, therefore the Hive queries h_{f1} and h_{f2} are connected using *UNION ALL*. Finally the SELECT clause is refined to include only variables, that are in *project* list of input query. The completely translated Hive query for *Triple store* and *Predicate store* storage schemes is shown in *listing:Translated Alternative Group Graph Pattern Hive Query.*

Translated Optional Group Graph Pattern Hive Query

```
Triple store: SELECT g1.var_recordID,g1.var_item
FROM (SELECT t1.subject as var_recordID,t1.object as
    var_item
  FROM TripleTable t1 JOIN TripleTable t2 ON(t1.
     object=t2.subject)
  WHERE t1.predicate= "mods:records" AND t2.
     predicate="rdf:type"
  AND t2.object="mods:Text")g1
LEFT OUTER JOIN (SELECT t3.subject as var_item, t3.
   object as
  var_version,t4.object as var_reference
  FROM TripleTable t3 JOIN TripleTable t4 ON(t3.
     subject=t4.subject)
  WHERE t3.predicate="mods:otherVersion" AND
  t4.predicate="mods:isReferencedBy")g2 ON(g1.
     var_item=g2.var_item);
Predicate store: SELECT g1.var_recordID,g2.
   var_version,g2.var_reference
FROM (SELECT t1.subject as var_recordID, t1.
   predicate["mods:records"] as var_item
  FROM PredicateTable t1 JOIN PredicateTable t2
  ON(t1.predicate["mods:records"]=t2.subject)
  WHERE t1.predicate["mods:records"] IS NOT NULL AND
  t2.predicate["rdf:type"] IS NOT NULL AND t2.
     predicate["rdf:type"]="mods:Text")g1
LEFT OUTER JOIN (SELECT t3.subject as var_item,t3.
   predicate["mods:otherVersion"] as
  var_version,t4.predicate["mods:isReferencedBy"] as
      var_reference
  FROM PredicateTable t3 JOIN PredicateTable t4 ON(
     t3.subject=t4.subject)
  WHERE t3.predicate["mods:otherVersion"] IS NOT
     NULL
  AND t4.predicate["mods:isReferencedBy"] IS NOT
     NULL)g2 ON(g1.var_item=g2.var_item);
```

Example 3: SPARQL Alternative Group Graph Pattern Query

```
SELECT ?item
Where { { ?item rdf:type mods:Text .
        ?item mods:subject ?object }
        UNION { ?item rdf:type mods:Text .
        ?subject mods:records ?item  } }
```

Translated Alternative Group Graph Pattern Hive Query

```
Triple store: SELECT g3.var_item
FROM (SELECT t1.subject as var_item
  FROM TripleTable t1 JOIN TripleTable t2 ON(t1.
     subject=t2.subject)
  WHERE t1.predicate="rdf:type" AND t1.object="mods:
     Text"
  AND t2.predicate="mods:subject"
UNION ALL
  SELECT t3.subject as var_item
  FROM TripleTable t3 JOIN TripleTable t4 ON(t3.
     subject=t4.object)
  WHERE t3.predicate="rdf:type" AND t3.object="mods:
     Text"
  AND t4.predicate="mods:records")g3;
Predicate store: SELECT g2.var_item
FROM (SELECT t1.subject as var_item
  FROM PredicateTable t1 JOIN PredicateTable t2 ON(
     t1.subject=t2.subject)
  WHERE t1.predicate["rdf:type"] IS NOT NULL AND t1.
     predicate["rdf:type"]="mods:Text"
  AND t2.predicate["mods:subject"] IS NOT NULL
UNION ALL
  SELECT t3.subject as var_item
  FROM PredicateTable t3 JOIN PredicateTable t4
  ON(t3.subject=t4.predicate["mods:records"])
  WHERE t3.predicate["rdf:type"] IS NOT NULL AND t3.
     predicate["rdf:type"]="mods:Text"
  AND t4.predicate["mods:records"] IS NOT NULL)g3;
```

Chapter 6

Evaluation and Results

The chapter evaluates presented translation and storage schema mapping algorithms.

6.1 Evaluation

The section describes the evaluation of the designed translation algorithm for the following features: correctness, support for different types of SPARQL queries (simple and complex), independence of translation from the storage structure. At the end of the section a features support comparison table of different scalable semantic data management solutions is also added. SPARQL-to-HiveQL translation layer enables data exploration in Hadoop based RDF stores, to prove this claim different experiments are conducted. The publicly available Barton data set is used for evaluation. This data is made from the RDF-formatted dump of the MIT libraries Barton catalog. It contains more than 50 million triples and there are 221 unique properties (Abadi et al., 2007).

Evaluation is performed using already defined performance evaluation criteria, mentioned in different research studies. The study (Chebotko et al., 2009) has defined *correctness*, *schema independence* and *efficiency* as a translation scheme evaluation criteria. Another study evaluated translation schemes on the basis of query *translation time* and *query transmission time* (Son et al., 2011). It has further added that translation time is affected by the number of translation algorithms, network environment as well as type of input query. Transmission time on the other hand is affected by the network environment, input query type, storage structure and data set size. Our main goal is to test the proposed translation algorithm for correctness, support for multiple SPARQL constructs and storage schema independence. Since optimization of query transmission time is not the key testing goal, so a relatively smaller segment of Barton data set will suffix the evaluation. The test data-set consists of 15,563 triples, 4,171 unique subjects, and 39 unique properties. The proposed translation algorithm translates an input SPARQL query into its Hive equivalent query. The translated queries are executed using Hive query engine on data stores built using HBase. Since newer versions of HBase are not compatible with Hadoop, therefore for testing purpose a Hadoop-0.20.2 append version is built. The Hadoop-0.20.2 append version, HBase-090.3 and Hive-0.9.0 version are configured to work together. The data repository: triple-table and predicate-table are created to store RDF data. In triple-table each triple is stored as a row and a unique auto increment unsigned long integer value is assigned as a row key. Therefore for the used Barton data set chunk, triple-table has 15, 563 rows. In predicate table, there are 4, 171 rows, where subject is the row key and object serves as a cell value. To simplify the evaluation process, we have implemented predicate-table approach with one table. The triple-table and predicate-tables are exposed in Hive as external tables, using the helper storage mapping Algorithms 4.4 and 4.7.

6.1.1 Test Queries Description

Test queries cover all categories of SPARQL queries that are: BGP, FGP, OGP and AGP. A short description of query meta-data is given below while the complete SPARQL queries are listed under appendix A.

- Q1-Q7: SPARQL Queries made of varying SPARQL constructs, including BGP with single triple pattern, BGP with multiple triple patterns,BGP with single triple pattern and Filter(F) constraint, BGP with multiple triple patterns and Filter constraint, OGP with single triple pattern OGP, OGP with multiple triple patterns and AGP/UGP with multiple BGPs).
- Q8-Q15: Complex SPARQL BGP queries including both bushy and property chain patterns and with a large number of variables.
- Q16-Q23: Complex Bushy pattern queries with 2 to 16 triple patterns.

In Figures Filter is refereed by F, OGP by O, AGP/UGP by U and Bushy pattern by B.

6.1.2 Evaluation for Correctness

Correctness is a fundamental and inevitable requirement for any translation process. Correctness needs to be checked at two levels. At first before the initiation of translation process, input must be validated, and then after the translation has been done. After query translation, it should be ensured, query semantics are preserved during the whole process. The coming sub-subsections includes a discussion on the aforementioned levels of correctness.

Correctness of Input Query

To ensure input query validation before the initiation of translation process, a set of unit test cases are designed and conducted using gUnit (gUnit Grammer Tool Website, 2012) Grammar Test Tool. The test cases are designed for all categories of SPARQL queries that are: BGP, FGP, OGP and AGP. Test cases are briefly summarised in table 6.1.

Correctness of the Translation algorithm

Correctness of the translation algorithm ensures that, semantics of an input query are not altered during the whole process. A translated query is correct if its results are complete and precise as compared to the input query results. To be more clear there should be no false positive or false negative in the solution set of the translated query. SPARQL queries Q1 to Q7 are executed using Apache Jena in-memory store. The solution set produced using Jena is considered as a benchmark solution set. The solution set produced by the same query after translation and then execution using Hive is compared to the

Table 6.1: Summarised Description of Unit Test Cases

Query Description	Category	Expected Result	Actual Result	gUnit Status	Remarks
BGP Queries	Positive Tests	Ok	Ok	✔	These test cases make sure that the parser accepts all valid BGP queries.
	Negative Tests	Fail	Fail	✔	These test cases make sure that the parser does not accepts any invalid BGP query.
FGP Queries	Positive Tests	Ok	Ok	✔	The set of designed test cases validates that all correct FGP queries are accepted by the parser.
	Negative Tests	Fail	Fail	✔	The set of designed test cases validates that all incorrect FGP queries are flagged as invalid.
OGP Queries	Positive Tests	Ok	Ok	✔	The test cases designed for this category make sure that the parser accepts only valid OGP queries.
	Negative Tests	Fail	Fail	✔	The test cases designed for this category make sure that the parser does not accept any invalid OGP query.
AGP/UGP Queries	Positive Tests	Ok	Ok	✔	Designed test cases make sure that the parser accepts only valid AGP queries.
	Negative Tests	Fail	Fail	✔	Designed test cases ensures that the parser properly flags grammatically invalid AGP queries.

solution set of Jena. The Comparison results are shown in table. 6.2. The solution sets produced by translated queries are equivalent to that of input queries. According to the observations, there are no false positive or false negative in the solution set of a translated query, therefore the translation algorithm is not modifying query semantics and hence is correct. It also proves the correctness of helper storage schema mapping procedures.

Table 6.2: Solution set comparison of Translated Hive Queries with Apache Jena

Query	Apache Jena	Hive	
	Triple-table Store	Triple-table Store	Predicate-table Store
Q1 (1 tp)	20	20	20
Q2 (2 tp)	288	288	288
Q3 (1 tp-F)	4165	4165	4165
Q4 (2 tp-F)	209	209	209
Q5 (2 tp-O)	299	299	299
Q6 (4 tp-O)	206	206	206
Q7 (4 tp-U)	1130	1130	1130

6.1.3 Efficient and Storage Independent Data Exploration Support for SPARQL

The sub-subsection describe the observations made during the execution of SPARQL queries with varying constructs on two storage anatomies: *Triple store* and *Predicate store*. The complexity of a SPARQL query can be measured along the following dimensions: varying SPARQL constructs, increasing number of triple patterns, increasing number of joins, increasing number of distinct variables. We tested the proposed translation algorithm along all mentioned axes of complexity. The observations and test results are discussed below.

Different SPARQL Constructs

To show working of the translation algorithm along this dimension, queries Q1 to Q7 (made of different SPARQL constructs) are translated. Query Q1 is an example of simplest SPARQL BGP query with only one triple pattern in its WHERE clause. It has been translated for both ontology independent triple-table and ontology dependent predicate-table storage approaches. Complexities of queries gradually grows from Q1 to Q7. The Fig. 6.1 shows the translation time of different queries. In Fig. 6.1 translation time is along Y-axis while queries are along X-axis. The Fig. 6.1 demonstrates the following observations 1) support for all categories of SPARQL queries and 2) translation time growth rate is minimal as compared to queries complexity and 3) there is no significant performance difference in translation time for triple-table and predicate-table storage approaches. The aforementioned observations prove the translation algorithm is storage schema independent and it supports various SPARQL query constructs.

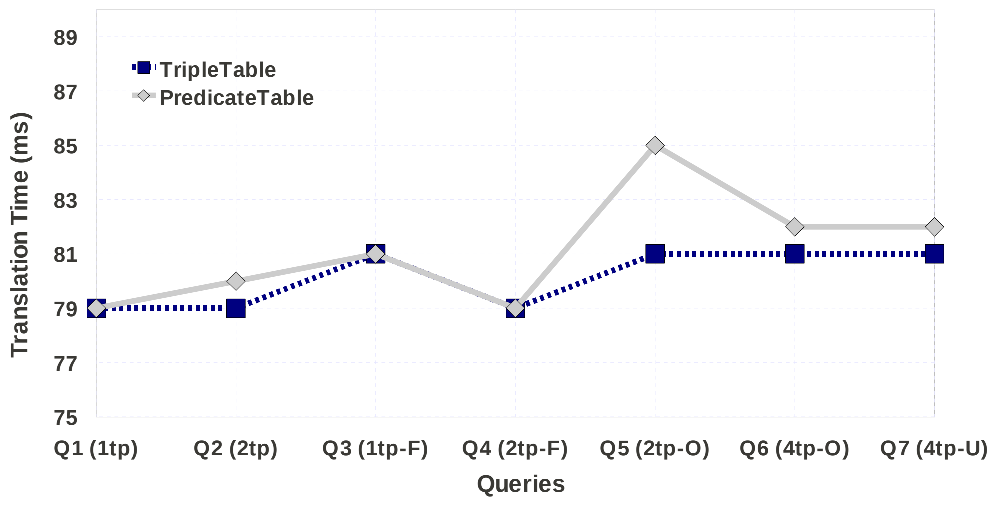

Figure 6.1: SPARQL Constructs Support

6.1.4 Complex SPARQL Queries

The number of participating triple patterns is a also a measure of query complexity. The increasing number of triple pattern implies increasing checks that must be performed on RDF data elements before allowing them to participate in solution set. The long property chain and Bushy pattern are common types of complex SPARQL queries, therefore the translation algorithm is evaluated for mentioned query types.

Property Chain Queries

To show the efficient translation support of proposed algorithm for Long property chain queries, queries Q8 to Q15 are translated. In Queries the number of triple pattern are increased as a multiple of 2. The increasing triple patterns add more variables, properties and joins to the input query. The translation of Q8 to Q15 is demonstrated in Fig. 6.2. The queries translation time is along Y-axis while queries (with an increasing number of triple patterns as a multiple of 2) are shown along X-axis. The Fig. 6.2 shows, translation time ranges from 80ms to 115ms for different queries starting with query Q8 (having 2 triple patterns) to Q15 (having 16 triple patterns). The comparison of query complexity with that of translation time pertains that Q15 is 8 times more complex than Q8, while the translation time is not even doubled. These observations prove efficient working of the translation algorithm for complex queries involving a large number of variables and property chains. The Fig. 6.2 shows, working of the translation algorithm for two different storage structures that are triple-table and predicate-table. Translation time curves in Fig. 6.2 are very close to each other thus proving efficient functionality of algorithm for different storage organizations.

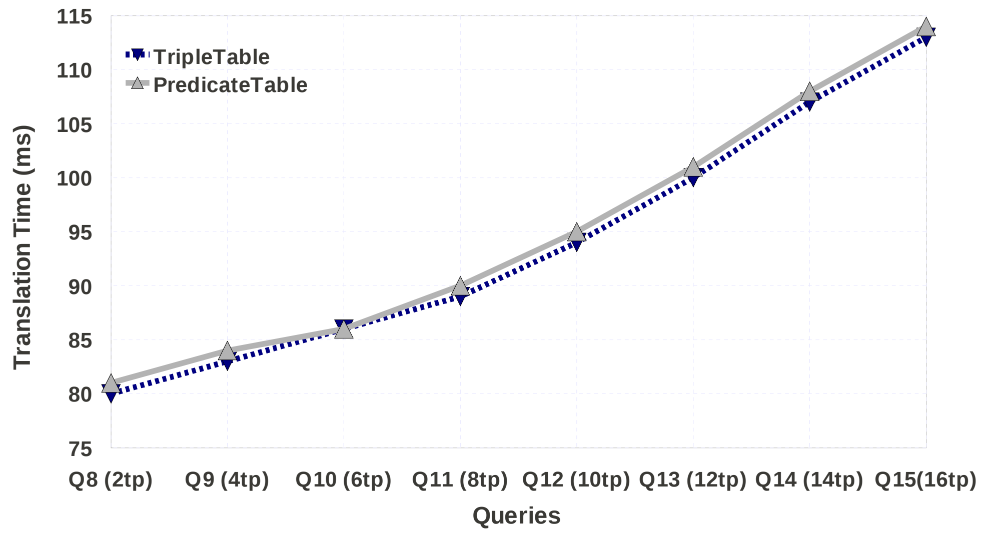

Figure 6.2: Complex Queries with Property Chain Patterns

Bushy Pattern Queries

To observe the translation algorithm behaviour along this dimension of increasing complexity, input queries Q16 (1 join) to Q23 (15 joins) are translated using proposed algorithm. The bushiness is increasing along each next query while starting from Q16 to the last test query Q23. The experiment results are shown in Fig. 6.3. In Fig. 6.3 Queries translation time is along Y-axis while queries with increasing complexity are along X-axis. Translation time curves for triple-table and predicate-table shows the time remains in range of 79ms to 93ms. The experiment proves the efficient working of proposed algorithm for complicated Bushy pattern queries with large number of joins.

6.1.5 Performance comparison of Flat versus Nested SPARQL Queries

According to literature query transmission time is dependent upon underlying storage structure, the number of translation algorithms, type of input query and size of data (Son et al., 2011). Type of translated query also impacts query transmission time and flat queries are many times faster than nested queries (Elliott et al., 2009b).

The proposed algorithm limits nesting to the number of group pattern in an input query, rather than number of triples patterns. Therefore generated queries are many times faster than their complete nested counter parts. The Hive decoded versions of Q1 to Q23 queries translated using proposed algorithm and their equivalent completely nested queries are tested for both triple-table and predicate-table storage schemes. The evaluation results are shown in Fig. 6.4. It has been observed that for simplest queries the performance of both flat and nested queries is same but with increasing query complexity the performance gape also grows rapidly. To conserve the space the Fig. 6.4 shows the queries Q1 to Q7

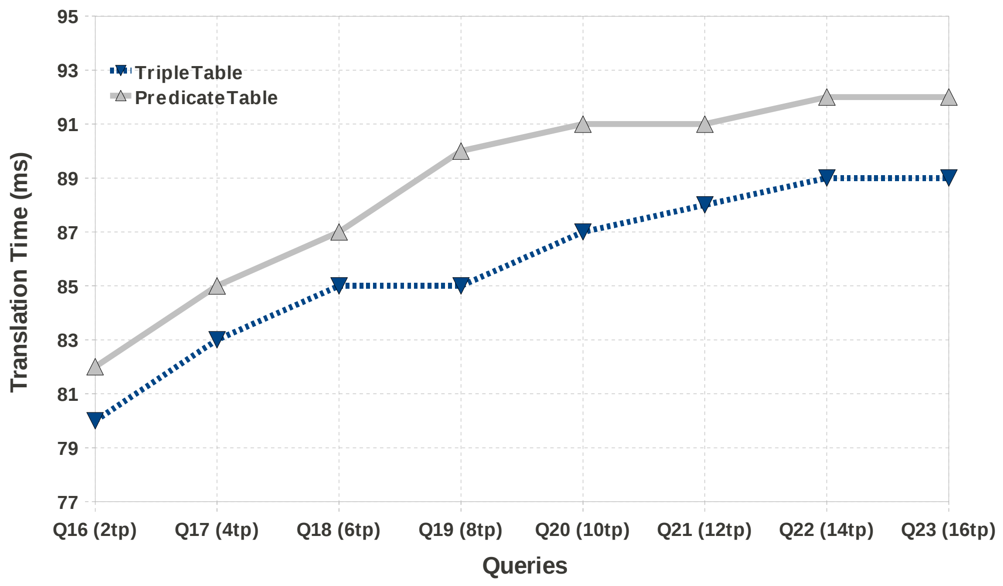

Figure 6.3: Complex Bushy Queries

made of different SPARQL constructs and Queries Q9 with 4 triple patterns, Q11 with 8 triple patterns and Q13 with 12 triple patterns. It is evident from the Fig. 6.4 that our generated queries are many times faster for both triple-table and predicate table storage structures as compared to their semantically equivalent completely nested counter parts. The other observation that is made in this experiment is that predicate-table storage structure decreases query transmission time.

6.1.6 Analysis of Scalable RDF Stores in Terms of Supported Features

To compare the proposed semantic data management solution with the other existing scalable RDF storage solutions (Hadoop based), a feature support summary is shown in table 6.3. The table clearly shows that our proposed Hadoop based RDF store with added SPARQL-to-HiveQL translation layer supersedes all other semantic data storage solutions in terms of supported features.

6.1.7 Evaluation Summary

The impact of underlying storage organization is minimal on translation time, while the complexity of input SPARQL queries mainly impacts the translation time. The evaluation done for long property chains and Bushy pattern queries has shown that the translation time is affected by the number of distinct variables and the number of Joins. Two queries

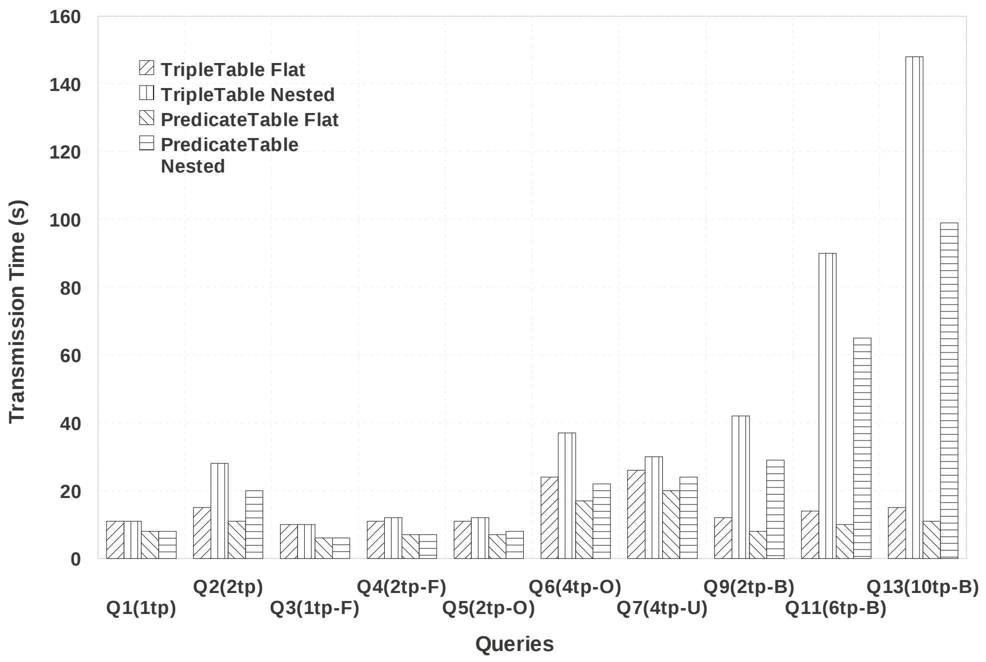

Figure 6.4: Flat vs Nested Queries Performance

with the same number of triple patterns and joins may take different amount of translation time if the number of variables in participating queries are different. The evaluation also shows that query transmission time can be decreased by limiting the level of nesting in decoded Hive queries.

Table 6.3: Feature Comparison table of different RDF stores

	SPARQL-to-HiveQL	Pig-Latin (Gates et al., 2009)	HBase (Franke et al., 2011; Sun and Jin, 2010)	Hadoop (Husain et al., 2010; Farhan Husain et al., 2009)	RDB (Chebotko et al., 2009; Elliott et al., 2009b; Lv et al., 2010; Son et al., 2008)
Scalability	✔	✔	✔	✔	✘
Flexible schema structure	✔	✔	✔	✘	✘
Rich set of data exploration commands	✔	✔	✘	✘	✔
Automatic Map-Reduce	✔	✔	✘	✘	✘
SPARQL BGP	✔	✔	✔	✔	✔
SPARQL BGP with Filter	✔	✔	✘	✔	✔
SPARQL OGP	✔	✘	✘	✘	✔
SPARQL AGP/UGP	✔	✘	✘	✘	✔
SPARQL Named Graphs	✘	✘	✘	✘	✔
Ontology dependent schema support	✔	✘	✔	✘	✔
Ontology independent schema support	✔	✘	✘	✔	✔
Completely flat queries	✘	✘	✘	✘	✔

Chapter 7

Conclusion and Future Work

The chapter summarises overall work done . It also discusses the dimensions in which this work could be further expanded.

7.1 Conclusion

In this thesis we formulated a scalable and efficient RDF data management solution using the distributed paradigm of Hadoop technologies. A storage schema independent translation algorithm for SPARQL-to-HiveQL translation is proposed. The translation algorithm is made independent of the underlying storage organization by delegating this functionality to the designed storage mapping procedure. The storage mapping procedure creates the schema view of the underlying storage and pass this information to the translation algorithm. Currently, the mapping algorithm can create virtual storage views for triple-table, hexa-table structure, property-table or even predicate-table storage organizations. Moreover, it can easily be extended to support future storage schemes built on top of Hadoop family of stores. The proposed translation algorithm is devised around the W3C defined SPARQL graph patterns. It supports complex SPARQL queries including BGP, FGP, OGP, AGP and GGP. Therefore it is capable of transparently translating SPARQL queries into semantically equivalent Hive queries, enabling RDF data querying on Hadoop based stores. The proposed algorithm has enabled RDF data querying using Hive engine thus realizing an efficient, scalable, fault tolerant and highly available RDF data management solution. The correctness and efficiency of the translation algorithm was tested using 23 different SPARQL queries of varying complexity on Barton data-set. The evaluation results show that the formulated approach retains support for complex queries without impacting translation efficiency on different storage layouts.

7.2 Future Work

In the future we would like to implement the proposed translation algorithm as map-reduce jobs. It will move the translation process closer to query execution and will allow query execution to be started as soon as parts of SPARQL queries get translated. The other possible expansions of work include adding support for the ASK, DESCRIBE, and CONSTRUCT queries.

Appendix A

SPARQL Test Queries

Query: 1

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?type
 Where { ?instances rdf:type ?type }
```

Query: 2

```
Query: 2
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mods:<http://simile.mit.edu/2006/01/ontologies/mods3#>
SELECT DISTINCT ?item
 Where { ?item rdf:type mods:Text .
 ?item mods:subject ?object }
```

Query: 3

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mods:<http://simile.mit.edu/2006/01/ontologies/mods3#>
SELECT ?item
 Where { ?item rdf:type ?type .
 FILTER(?type != mods:NotatedMusic) }
```

Query: 4

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mods:<http://simile.mit.edu/2006/01/ontologies/mods3#>
SELECT ?recordID
 Where { ?recordID mods:records ?item .
 ?item rdf:type ?type .
 FILTER( ?type!= mods:StillImage ) }
```

Query: 5

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mods:<http://simile.mit.edu/2006/01/ontologies/mods3#>
SELECT ?recordID
 Where { ?recordID mods:records ?item .
 OPTIONAL { ?item rdf:type ?type }      }
```

Query: 6

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mods:<http://simile.mit.edu/2006/01/ontologies/mods3#>
PREFIX role:<http://simile.mit.edu/2006/01/role/>
SELECT Reduced ?recordID
 Where { ?recordID mods:records ?item .
 ?item rdf:type mods:Text .
 OPTIONAL { ?item mods:otherVersion ?version .
 ?item mods:isReferencedBy ?reference  } }
```

Query: 7

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mods:<http://simile.mit.edu/2006/01/ontologies/mods3#>
SELECT ?item
 Where { { ?item rdf:type mods:Text .
 ?item mods:subject ?object  }
 UNION { ?item rdf:type mods:Text .
 ?subject mods:records ?item } }
```

Query: 8

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mods:<http://simile.mit.edu/2006/01/ontologies/mods3#>
SELECT ?recordID
Where { ?recordID mods:records ?item .
 ?item rdf:type mods:Text }
```

Query: 9

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mods:<http://simile.mit.edu/2006/01/ontologies/mods3#>
SELECT ?recordID
 Where { ?recordID mods:records ?item .
 ?item rdf:type mods:Text .
 ?item mods:genre ?genre .
 ?item mods:classification ?classification }
```

Query: 10

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mods:<http://simile.mit.edu/2006/01/ontologies/mods3#>
PREFIX role:<http://simile.mit.edu/2006/01/role/>
SELECT ?recordID
 Where { ?recordID mods:records ?item .
 ?item rdf:type mods:Text .
 ?item mods:genre ?genre .
 ?item mods:classification ?classification .
 ?item mods:subject ?subject .
 ?item role:creator ?creator  }
```

Query: 11

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mods:<http://simile.mit.edu/2006/01/ontologies/mods3#>
PREFIX role:<http://simile.mit.edu/2006/01/role/>
SELECT ?recordID
 Where { ?recordID mods:records ?item .
 ?item rdf:type mods:Text .
 ?item mods:genre ?genre .
 ?item mods:classification ?classification .
 ?item mods:subject ?subject .
 ?item role:creator ?creator .
 ?item mods:language ?language .
 ?item mods:contents ?contents }
```

Query: 12

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mods:<http://simile.mit.edu/2006/01/ontologies/mods3#>
PREFIX role:<http://simile.mit.edu/2006/01/role/>
SELECT ?recordID
 Where { ?recordID mods:records ?item .
 ?item rdf:type mods:Text .
 ?item mods:genre ?genre .
 ?item mods:classification ?classification .
 ?item mods:subject ?subject .
 ?item role:creator ?creator .
 ?item mods:language ?language .
 ?item mods:contents ?contents .
 ?item mods:note ?note .
 ?item mods:dateIssued ?dateIssued }
```

Query: 13

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mods:<http://simile.mit.edu/2006/01/ontologies/mods3#>
PREFIX role:<http://simile.mit.edu/2006/01/role/>
SELECT ?recordID
 Where { ?recordID mods:records ?item .
 ?item rdf:type mods:Text .
 ?item mods:genre ?genre .
 ?item mods:classification ?classification .
 ?item mods:subject ?subject .
 ?item role:creator ?creator .
 ?item mods:language ?language .
 ?item mods:contents ?contents .
 ?item mods:note ?note .
 ?item mods:dateIssued ?dateIssued .
 ?item mods:relatedTo ?relatedTo .
 ?item mods:title ?title }
```

Query: 14

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mods:<http://simile.mit.edu/2006/01/ontologies/mods3#>
PREFIX role:<http://simile.mit.edu/2006/01/role/>
SELECT ?recordID
 Where { ?recordID mods:records ?item .
 ?item rdf:type mods:Text .
 ?item mods:genre ?genre .
 ?item mods:classification ?classification .
 ?item mods:subject ?subject .
 ?item role:creator ?creator .
 ?item mods:language ?language .
 ?item mods:contents ?contents .
 ?item mods:note ?note .
 ?item mods:dateIssued ?dateIssued .
 ?item mods:relatedTo ?relatedTo .
 ?item mods:title ?title .
 ?item mods:edition ?edition .
 ?item mods:publisher ?publisher }
```

Query: 15

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mods:<http://simile.mit.edu/2006/01/ontologies/mods3#>
PREFIX role:<http://simile.mit.edu/2006/01/role/>
SELECT ?recordID
 Where { ?recordID mods:records ?item .
 ?item rdf:type mods:Text .
 ?item mods:genre ?genre .
 ?item mods:classification ?classification .
 ?item mods:subject ?subject .
 ?item role:creator ?creator .
 ?item mods:language ?language .
 ?item mods:contents ?contents .
 ?item mods:note ?note .
 ?item mods:dateIssued ?dateIssued .
 ?item mods:relatedTo ?relatedTo .
 ?item mods:title ?title .
 ?item mods:edition ?edition .
 ?item mods:publisher ?publisher .
 ?publisher mods:location ?location .
 ?location mods:name ?name }
```

Query:16

```
PREFIX mods:<http://simile.mit.edu/2006/01/ontologies/mods3#>
PREFIX publisher: <http://simile.mit.edu/2006/01/publisher/>
SELECT ?recordID
 Where { ?recordID mods:publisher publisher:
    Temple_University_Press ;
 mods:subject place:Pennsylvania ; }
```

Query: 17

```
PREFIX mods:<http://simile.mit.edu/2006/01/ontologies/mods3#>
PREFIX role:<http://simile.mit.edu/2006/01/role/>
PREFIX topic:<http://simile.mit.edu/2006/01/topic/>
PREFIX publisher: <http://simile.mit.edu/2006/01/publisher/>
PREFIX place:<http://simile.mit.edu/2006/01/place/>
SELECT ?recordID
 Where { ?recordID mods:publisher publisher:
    Temple_University_Press ;
 mods:subject place:Pennsylvania ;
 role:creator  <http://simile.mit.edu/2006/01/entity#Blumberg
    ,_Leonard_U._1920-> ;
 mods:subject topic:Alcoholism }
```

Query: 18

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mods:<http://simile.mit.edu/2006/01/ontologies/mods3#>
PREFIX role:<http://simile.mit.edu/2006/01/role/>
PREFIX topic:<http://simile.mit.edu/2006/01/topic/>
PREFIX publisher: <http://simile.mit.edu/2006/01/publisher/>
PREFIX place:<http://simile.mit.edu/2006/01/place/>
SELECT ?recordID
 Where { ?recordID mods:publisher publisher:
    Temple_University_Press ;
 mods:subject place:Pennsylvania ;
 role:creator  <http://simile.mit.edu/2006/01/entity#Blumberg
    ,_Leonard_U._1920-> ;
 mods:subject topic:Alcoholism ;
 mods:subject place:Philadelphia ;
 role:creator <http://simile.mit.edu/2006/01/entity#Shandler,
    _Irving_W.> }
```

Query: 19

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mods:<http://simile.mit.edu/2006/01/ontologies/mods3#>
PREFIX role:<http://simile.mit.edu/2006/01/role/>
PREFIX topic:<http://simile.mit.edu/2006/01/topic/>
PREFIX publisher: <http://simile.mit.edu/2006/01/publisher/>
PREFIX place:<http://simile.mit.edu/2006/01/place/>
SELECT ?recordID
 Where { ?recordID mods:publisher publisher:
    Temple_University_Press ;
 mods:subject place:Pennsylvania ;
 role:creator  <http://simile.mit.edu/2006/01/entity#Blumberg
    ,_Leonard_U._1920-> ;
 mods:subject topic:Alcoholism ;
 mods:subject place:Philadelphia ;
 role:creator <http://simile.mit.edu/2006/01/entity#Shandler,
    _Irving_W.> ;
 rdf:type mods:Text ;
 role:creator <http://simile.mit.edu/2006/01/entity#Shipley,
    _Thomas_E.> }
```

Query: 20

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mods:<http://simile.mit.edu/2006/01/ontologies/mods3#>
PREFIX role:<http://simile.mit.edu/2006/01/role/>
PREFIX topic:<http://simile.mit.edu/2006/01/topic/>
PREFIX publisher: <http://simile.mit.edu/2006/01/publisher/>
PREFIX place:<http://simile.mit.edu/2006/01/place/>
SELECT ?recordID
 Where { ?recordID mods:publisher publisher:
    Temple_University_Press ;
 mods:subject place:Pennsylvania ;
 role:creator  <http://simile.mit.edu/2006/01/entity#Blumberg
    ,_Leonard_U._1920-> ;
 mods:subject topic:Alcoholism ;
 mods:subject place:Philadelphia ;
 role:creator <http://simile.mit.edu/2006/01/entity#Shandler,
    _Irving_W.> ;
 rdf:type mods:Text ;
 role:creator <http://simile.mit.edu/2006/01/entity#Shipley,
    _Thomas_E.> ;
 mods:note "[by] Leonard Blumberg, Thomas E. Shipley, Jr., [
    and] Irving W. Shandler." ;
 mods:language <http://simile.mit.edu/2006/01/language/iso639
    -2b/eng> }
```

Query: 21

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mods:<http://simile.mit.edu/2006/01/ontologies/mods3#>
PREFIX role:<http://simile.mit.edu/2006/01/role/>
PREFIX topic:<http://simile.mit.edu/2006/01/topic/>
PREFIX publisher: <http://simile.mit.edu/2006/01/publisher/>
PREFIX place:<http://simile.mit.edu/2006/01/place/>
SELECT ?recordID
 Where { ?recordID mods:publisher publisher:
    Temple_University_Press ;
 mods:subject place:Pennsylvania ;
 role:creator  <http://simile.mit.edu/2006/01/entity#Blumberg
    ,_Leonard_U._1920-> ;
 mods:subject topic:Alcoholism ;
 mods:subject place:Philadelphia ;
 role:creator <http://simile.mit.edu/2006/01/entity#Shandler,
    _Irving_W.> ;
 rdf:type mods:Text ;
 role:creator <http://simile.mit.edu/2006/01/entity#Shipley,
    _Thomas_E.> ;
 mods:note "[by] Leonard Blumberg, Thomas E. Shipley, Jr., [
    and] Irving W. Shandler." ;
 mods:language <http://simile.mit.edu/2006/01/language/iso639
    -2b/eng> ;
 mods:subject topic:Drinking_of_alcoholic_beverages ;
 mods:subject topic:Treatment }
```

Query: 22

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mods:<http://simile.mit.edu /2006/01/ontologies/mods3
    #>
PREFIX role:<http://simile.mit.edu/2006/01/role/>
PREFIX topic:<http://simile.mit.edu/2006/01/topic/>
PREFIX publisher: <http://simile.mit.edu/2006/01/publisher/>
PREFIX place:<http://simile.mit.edu/2006/01/place/>
SELECT ?recordID
 Where { ?recordID mods:publisher publisher:
    Temple_University_Press ;
 mods:subject place:Pennsylvania ;
 role:creator  <http://simile.mit.edu/2006/01/entity#Blumberg
    ,_Leonard_U._1920-> ;
 mods:subject topic:Alcoholism ;
 mods:subject place:Philadelphia ;
 role:creator <http://simile.mit.edu/2006/01/entity#Shandler,
    _Irving_W.> ;
 rdf:type mods:Text ;
 role:creator <http://simile.mit.edu/2006/01/entity#Shipley,
    _Thomas_E.> ;
 mods:note "[by] Leonard Blumberg, Thomas E. Shipley, Jr., [
    and] Irving W. Shandler." ;
 mods:language <http://simile.mit.edu/2006/01/language/iso639
    -2b/eng> ;
 mods:subject topic:Drinking_of_alcoholic_beverages ;
 mods:subject topic:Treatment ;
 mods:note "Bibliography: p. [289]-297." ;
 mods:subject topic:Alcoholics }
```

Query: 23

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mods:<http://simile.mit.edu/2006/01/ontologies/mods3#>
PREFIX role:<http://simile.mit.edu/2006/01/role/>
PREFIX topic:<http://simile.mit.edu/2006/01/topic/>
PREFIX publisher: <http://simile.mit.edu/2006/01/publisher/>
PREFIX place:<http://simile.mit.edu/2006/01/place/>
SELECT ?recordID
 Where { ?recordID mods:publisher publisher:
    Temple_University_Press ;
 mods:subject place:Pennsylvania ;
 role:creator  <http://simile.mit.edu/2006/01/entity#Blumberg
    ,_Leonard_U._1920-> ;
 mods:subject topic:Alcoholism ;
 mods:subject place:Philadelphia ;
 role:creator <http://simile.mit.edu/2006/01/entity#Shandler,
    _Irving_W.> ;
 rdf:type mods:Text ;
 role:creator <http://simile.mit.edu/2006/01/entity#Shipley,
    _Thomas_E.> ;
 mods:note "[by] Leonard Blumberg, Thomas E. Shipley, Jr., [
    and] Irving W. Shandler." ;
 mods:language <http://simile.mit.edu/2006/01/language/iso639
    -2b/eng> ;
 mods:subject topic:Drinking_of_alcoholic_beverages ;
 mods:subject topic:Treatment ;
 mods:note "Bibliography: p. [289]-297." ;
 mods:subject topic:Alcoholics ;
 mods:subject topic:Slums ;
 mods:issuance "monographic" }
```

Bibliography

Abadi, D., Marcus, A., Madden, S., and Hollenbach, K. (2007). Using the barton libraries dataset as an rdf benchmark. Technical report, Technical Report MIT-CSAIL-TR-2007-036, MIT.

Abadi, D., Marcus, A., Madden, S., and Hollenbach, K. (2009). Sw-store: a vertically partitioned dbms for semantic web data management. *The VLDB Journal*, 18(2):385–406.

AllegroGraph (2012). Allegrograph. http://www.franz.com/agraph/allegrograph/. (Last visited: June 2012).

ANTLR Website (2012). Antlr. http://www.antlr.org/. (Last visited: October 2012).

Apache Hadoop Website (2012). Apache hadoop. http://hadoop.apache.org. (Last visited: June 2012).

Apache HBase Website (2012a). Apache hbase. http://hbase.apache.org. (Last visited: June 2012).

Apache HBase Website (2012b). Apache hbase refrence guide. http://hbase.apache.org/book.html#performance. (Last visited: July 2012).

Apache Hive Website (2012). Apache hive. http://hive.apache.org. (Last visited: November 2011).

Bajda-Pawlikowski, K. (2008). Querying rdf data stored in dbms: Sparql to sql conversion. Technical report, Technical Report TR-1409, Yale Computer Science Department, USA.

Bizer, C., Heath, T., and Berners-Lee, T. (2009a). Linked data-the story so far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5:1–22.

Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., and Hellmann, S. (2009b). Dbpedia-a crystallization point for the web of data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):154–165.

Broekstra, J., Kampman, A., and Van Harmelen, F. (2002). Sesame: A generic architecture for storing and querying rdf and rdf schema. *The Semantic Web, ISWC 2002*, pages 54–68.

Chebotko, A., Lu, S., and Fotouhi, F. (2009). Semantics preserving sparql-to-sql translation. *Data & Knowledge Engineering*, 68(10):973–1000.

Chebotko, A., Lu, S., Jamil, H., and Fotouhi, F. (2006). Semantics preserving sparql-to-sql query translation for optional graph patterns. Technical report, Technical Report TR-DB-052006-CLJF.

Chong, E., Das, S., Eadon, G., and Srinivasan, J. (2005). An efficient sql-based rdf querying scheme. In *Proceedings of the 31st international conference on Very large data bases*, pages 1216–1227. VLDB Endowment.

Cyganiak, R. (2005). A relational algebra for sparql. *Digital Media Systems Laboratory HP Laboratories Bristol. HPL-2005-170.*

Dean, J. and Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Dyer, C., Cordova, A., Mont, A., and Lin, J. (2008). Fast, easy, and cheap: construction of statistical machine translation models with mapreduce. In *Proceedings of the Third Workshop on Statistical Machine Translation*, pages 199–207. Association for Computational Linguistics.

Elliott, B., Cheng, E., Thomas-Ogbuji, C., and Ozsoyoglu, Z. (2009a). A complete translation from sparql into efficient sql. In *Proceedings of the 2009 International Database Engineering & Applications Symposium*, pages 31–42. ACM.

Elliott, B., Cheng, E., Thomas-Ogbuji, C., and Ozsoyoglu, Z. (2009b). A complete translation from sparql into efficient sql. In *Proceedings of the 2009 International Database Engineering & Applications Symposium*, pages 31–42. ACM.

Erling, O. (2001). Implementing a sparql compliant rdf triple store using a sql-ordbms. Technical report, Technical Report, OpenLink Software Virtuoso.

Farhan Husain, M., Doshi, P., Khan, L., and Thuraisingham, B. (2009). Storage and retrieval of large rdf graph using hadoop and mapreduce. *Cloud Computing*, pages 680–686.

Franke, C., Morin, S., Chebotko, A., Abraham, J., and Brazier, P. (2011). Distributed semantic web data management in hbase and mysql cluster. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 105–112. IEEE.

Freebase (2012). Freebase. http://www.freebase.com. (Last visited: June 2012).

Gates, A. F., Natkovich, O., Chopra, S., Kamath, P., Narayanamurthy, S. M., Olston, C., Reed, B., Srinivasan, S., and Srivastava, U. (2009). Building a high-level dataflow system on top of map-reduce: the pig experience. *Proc. VLDB Endow.*, 2(2):1414–1425.

gUnit Grammer Tool Website (2012). gunit. http://www.antlr.org/wiki/display/ANTLR3/gUnit+-+Grammar+Unit+Testing. (Last visited: October 2012).

Guo, Y., Pan, Z., and Heflin, J. (2005). Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182.

Hadoop Wiki Website (2012). Hadoop wiki. http://wiki.apache.org/hadoop/PoweredBy. (Last visited: June 2012).

Harris, S. and Gibbins, N. (2003). 3store: Efficient bulk rdf storage. In Volz, R., Decker, S., and Cruz, I., editors, *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PSSS)*, volume 89. CEUR-WS.

Harris, S. and Shadbolt, N. (2005). Sparql query processing with conventional relational database systems. In *Web Information Systems Engineering–WISE 2005 Workshops*, pages 235–244. Springer.

Husain, M., Khan, L., Kantarcioglu, M., and Thuraisingham, B. (2010). Data intensive query processing for large rdf graphs using cloud computing tools. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 1–10. IEEE.

Jia, Y. and Shao, Z. (2009). Hive performance benchmarks. Feature Report HIVE-396, Apache Software Foundation. https://issues.apache.org/jira/browse/HIVE-396.

Kumar, V., Andrade, H., Gedik, B., and Wu, K. (2010). Deduce: at the intersection of mapreduce and stream processing. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 657–662. ACM.

Lee, R., Luo, T., Huai, Y., Wang, F., He, Y., and Zhang, X. (2011). Ysmart: Yet another sql-to-mapreduce translator. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 25–36. IEEE.

Lee, T., Hendler, J., Lassila, O., et al. (2001). The semantic web. *Scientific American*, 284(5):34–43.

Lin, J. and Dyer, C. (2010). Data-intensive text processing with mapreduce. *Synthesis Lectures on Human Language Technologies*, 3(1):1–177.

Lu, J., Cao, F., Ma, L., Yu, Y., and Pan, Y. (2008). An effective sparql support over relational databases. *Semantic Web, Ontologies and Databases*, pages 57–76.

Lv, L., Jiang, H., and Ju, L. (2010). Research and implementation of the sparql-to-sql query translation based on restrict rdf view. In *Web Information Systems and Mining (WISM), 2010 International Conference on*, volume 1, pages 309–313. IEEE.

Manola, F., Miller, E., and McBride, B. (2004). Rdf primer. W3c recommendation, World Wide Web Consortium. http://www.w3.org/TR/rdf-primer/.

Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. (2008). Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM.

Prud Hommeaux, E. and Seaborne, A. (2008). Sparql query language for rdf. W3c recommendation, World Wide Web Consortium.

Sakr, S. and Al-Naymat, G. (2010). Relational processing of rdf queries: a survey. *SIGMOD Rec.*, 38(4):23–28.

Son, J., Jeong, D., and Baik, D. (2008). Practical approach: Independently using sparql-to-sql translation algorithms on storage. In *Networked Computing and Advanced Information Management, 2008. NCM'08. Fourth International Conference on*, volume 2, pages 598–603. IEEE.

Son, J., Kim, J., and Baik, D. (2011). Performance evaluation of storage-independent model for sparql-to-sql translation algorithms. In *New Technologies, Mobility and Security (NTMS), 2011 4th IFIP International Conference on*, pages 1–4. IEEE.

Sridhar, R., Ravindra, P., and Anyanwu, K. (2009). Rapid: Enabling scalable ad-hoc analytics on the semantic web. *The Semantic Web-ISWC 2009*, pages 715–730.

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

Theoharis, Y., Christophides, V., and Karvounarakis, G. (2005). Benchmarking database representations of rdf/s stores. *The Semantic Web–ISWC 2005*, pages 685–701.

Thusoo, A., Sarma, J., Jain, N., Shao, Z., Chakka, P., Zhang, N., Antony, S., Liu, H., and Murthy, R. (2010). Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE.

W3C Website (2012). World wide web consortium (w3c). http://www.w3.org/. (Last visited: October 2012).

WebDataCommons (2012). Webdatacommons. http://webdatacommons.org. (Last visited: June 2012).

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

Wilkinson, K., Sayers, C., Kuno, H., Reynolds, D., et al. (2003). Efficient rdf storage and retrieval in jena2. In *Proceedings of SWDB*, volume 3, pages 131–150.

Zhang, C., De Sterck, H., Aboulnaga, A., Djambazian, H., and Sladek, R. (2010). Case study of scientific data processing on a cloud using hadoop. In *High Performance Computing Systems and Applications*, pages 400–415. Springer.

Zhou, C. and Zheng, Y. (2011). Query rewriting from sparql to sql for relational database integration. *IEIT Journal of Adaptive & Dynamic Computing*, 1(1):1–8.