The Pennsylvania State University
The Graduate School

RDF3X-MPI: A PARTITIONED RDF ENGINE FOR DATA-PARALLEL SPARQL QUERYING

A Thesis in
Computer Science and Engineering
by
Sai Krishnan Chirravuri

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2014

The thesis of Sai Krishnan Chirravuri was reviewed and approved* by the following:

Kamesh Madduri
Assistant Professor of Computer Science and Engineering
Thesis Advisor

Piotr Berman
Associate Professor of Computer Science and Engineering

Lee Coraor
Associate Professor of Computer Science and Engineering
Director of Academic Affairs of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

The Semantic Web is a collection of technologies that facilitate universal access to linked data. The Resource Description Framework (RDF) model is one such technology that is being developed by the World Wide Web Consortium (W3C). A common representation of RDF data is as a set of triples. Each triple contains three fields: a subject, a predicate, and an object. A collection of triples can also be visualized as a directed graph, with subjects and objects as vertices in the graph, and predicates as edges connecting the vertices. When large collections of triples are aggregated, they form massive RDF graphs. Collections of RDF triple data sets have been growing over the past decade, and publicly-available RDF data sets now have billions of triples. As data sizes continue to grow, the time to process and query large RDF data sets also continues to increase. This work presents RDF3X-MPI, a new scalable, parallel RDF data management and querying system based on the RDF-3X data management system. RDF-3X (RDF Triple eXpress) is a state-of-the-art RDF engine that is shown to outperform alternatives by one or two orders of magnitude, on several well-known benchmarks and in experimental studies. Our approach leverages all the data storage, indexing, and querying optimizations in RDF-3X. We additionally partition input RDF data to support parallel data ingestion, and devise a methodology to execute SPARQL queries in parallel, with minimal inter-processor communication. Using our new approach, we demonstrate a performance improvement of up to 12.9× in query evaluation for the LUBM benchmark, using 32-way MPI task parallelism. This work also presents an in-depth characterization of SPARQL query execution times with RDF-3X and RDF3X-MPI on several large-scale benchmark instances.

Table of Contents

List of Figures

List of Tables

Acknowledgments

Foremost, I would like to express my sincere gratitude to my advisor, Dr. Kamesh Madduri for his continuous guidance on this project. My sincere thanks also goes to the authors of RDF-3x Thomas Neumann and Gerhard Weikum for replying to my emails about the intricate details of their codebase. I also thank my lab mate Shad Kirmani for his encouraging words. Finally, I thank my friends Praveen Yedlapally, Bhanu Kishore and Rahul BMV for the stimulated discussion and the fun we had while discussing several topics related to my research.

Dedication

To my family.

Chapter 1

Introduction

The Semantic Web refers to various technologies that make the vast knowledge present on the worldwide web readable by machines. One of the goals of the Semantic Web is to make data from a wide variety of sources available under a common and unified standard. Data can then be shared across different domains, enabling applications to access, query and understand the content. Many languages have been developed to describe the information on web resources. The Resource Description Framework (RDF) [1] is a technology included in the Semantic Web as recommended by the World Wide Web Consortium (W3C). It consists of triples of the form subject-predicate-object. An RDF data set can also be visualized as a graph, with subjects and objects forming vertices in the graph, and predicates representing edges connecting the vertices. Yago [2], Dbpedia [3], Uniprot [4] are some of the initiatives that provide large-scale RDF knowledge stores. The SPARQL Protocol and RDF Query Language (SPARQL) [5] is a specialized query language used for querying RDF data. SPARQL has been recommended by the W3C and is considered a key semantic web technology.

1.1 Problem Statement

Specialized solutions exist for efficiently storing, indexing, and querying of RDF data sets. These solutions are referred to as RDF stores, triple stores, or RDF engines. With the explosion of data sets in the RDF format, the volume of RDF data is growing beyond the processing capacity of conventional single-server RDF

stores. In the past few years, several efficient and high-performance RDF stores for sequential processing have been developed. RDF-3X is one such prominent RDF store with support for fast SPARQL query processing. While engines such as RDF-3X perform well on single server, they are not designed to run in a distributed environment. Current distributed-memory clusters and parallel systems have several terabytes of aggregate main memory. An RDF store that can exploit such a compute environment would be able to process RDF data with with tens of billions of triple sets. Utilizing parallel systems will also enable us to improve query processing time through parallelism. Thus, the goal of this thesis is to design a scalable RDF data processing engine on a distributed system, utilizing an efficient RDF data store in a single-server configuration.

1.2 Proposed Solution

We propose a scalable engine RDF3X-MPI to process RDF data by extending the open-source RDF store RDF-3X. The key performance determinants for SPARQL query processing using an RDF engine are the indexing methodology employed, the design of the SPARQL query processing engine, support for query optimization, and the query evaluation engine. RDF-3X employs a novel architecture with innovations in each of these key components. We extend RDF-3X for efficient execution on distributed-memory systems, using the Message Passing Interface (MPI) [6] library for communication and coordination. Specifically, we develop two utilities, RDF3X-MPI*load* and RDF3X-MPI*query*, for parallel loading/indexing RDF data sets and parallel querying, respectively. We also evaluate parallel load and query performance using the three benchmarks on a cluster system. On a synthetic LUBM data set generated using universities count set to 1000 (LUBM1000) and with 32-way parallelism, RDF3X-MPI achieves speedup upto 12.9×. We observe that the resulting speedup is proportional to the size of the output generated by a query.

1.3 Thesis Organization

The rest of this thesis discusses the key concepts involved in designing a partitioned RDF engine for data-parallel SPARQL querying. The thesis chapters are organized as follows. Chapter 2 focuses on introducing terms used in this work, in order to familiarize the reader with the technologies we are working on. Chapter 3 gives a detailed description of the state-of-the-art RDF-3X. Chapter 4 presents details about our implementation of the load step of our distributed data store. Chapter 5 complements the previous chapter by describing the data-parallel execution of a query. Chapter 6 describes the experimental setup and presents performance results from this work. Chapter 7 concludes this thesis by summarizing the implementation and key results. It also outlines future work towards developing this distributed RDF store.

Chapter 2

Background

The Semantic Web [7, 8] is a concept introduced by Tim Berners-Lee as a component in Web 3.0. It constitutes a set of technologies used to give meaning to the information available on the web. The aim of the Semantic Web is to enable computers to understand information on web resources. The Semantic Web has become very popular. It is now being used in several applications in the fields of electronic commerce, social networking, as well as computational science.

2.1 RDF

The Resource Description Framework (RDF) data model is one of the most common formats for representing unstructured semantic data. Originally introduced in 1998, the current version of RDF was standardized by the W3C in February of 2004. RDF is the result of one of the goals set by the Semantic Web program: to develop a language for describing metadata in a distributed, extensible setting [9]. An ontology [10, 11] is a model of the RDF data, listing the types of objects, the relationships that connect them, and constraints on the ways that objects and relationships can be combined. Having said that, the use of RDF does not require the presence of an explicit ontology or schema, making it a flexible and highly extensible language. Data sets available in the RDF format are growing by the day. Several public-domain data sets extract information from Wikipedia like Freebase and DBpedia. UniProt is an example of a more specialized data set that provides information on protein sequences. These data sets are rapidly growing and the

larger publicly-available ones now have billions of triples.

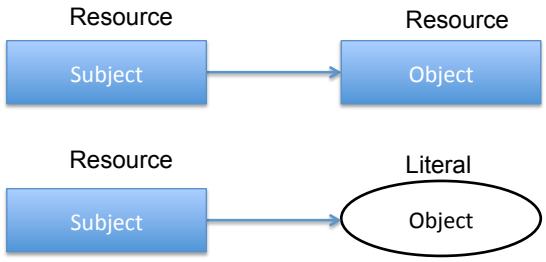


Figure 2.1: Visual representation of subjects and objects in RDF triples.

In the RDF model, data records are organized as ⟨subject, predicate, object⟩ expressions known as triples. A predicate describes the relationship between a subject and an object. An RDF data set can also be visualized as a graph with subjects and objects forming vertices or nodes in the graph, and predicates representing edges connecting two vertices The subject, predicate, and object entities in an RDF graph can either be Uniform Resource Identifiers (URIs) or Blank nodes. Objects can additionally be of the literal type [12]. In Figure 2.1, literals are denoted by an oval box and other types of subjects and objects are denoted by the rectangular box. Table 2.1 lists sample RDF triples from a data set with Abraham Lincoln as the subject, and Figure 2.2 shows an example graph visualization of an RDF data set.

Table 2.1: RDF *triples* with y:Abraham_Lincoln as the subject and prefix y: http://en.wikipedia.org/wiki/.

Subject	Predicate	Object
y:Abraham_Lincoln	hasName	"Abraham Lincoln"
y:Abraham_Lincoln	gender	"Male"
y:Abraham_Lincoln	title	"President"
y:Abraham_Lincoln	bornOnDate	"1809-02-12"
y:Abraham_Lincoln	diedOnDate	"1865-04-15"
y:Abraham_Lincoln	diedIn	y:Washington_D.C.
y:Abraham_Lincoln	bornIn	y:Hodgenville_KY

Notation-3, Terse RDF Triple language (Turtle), N-triples, RDF/XML are some of the popular serialization formats for the RDF data. Compared to the rest of them, XML is extremely well-known and widely accepted standard. Standard

library library parsers for XML are readily available in virtually every programming language. This makes XML more preferable from a purely interoperability standpoint.

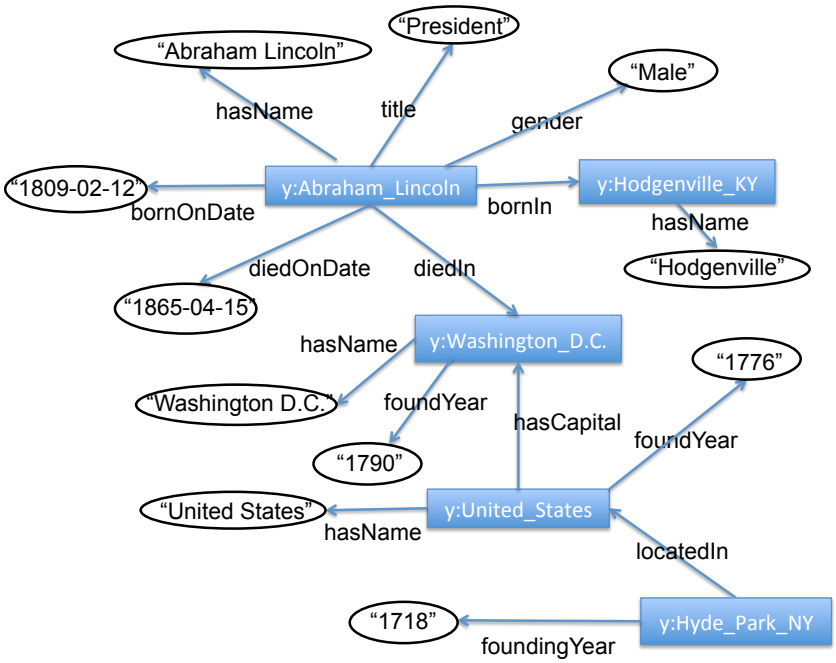


Figure 2.2: An example RDF graph constructed from triples in Table 2.1.

2.2 RDF Stores

There are several existing and specialized solutions for efficiently storing, indexing, and querying RDF data sets. These solutions are referred to as RDF stores, triple stores, or RDF engines. Virtuoso [13], Jena [14], Sesame [15], Stardog [16], 4store [17], AllegroGraph [18] are some widely-used commercial and open-source RDF engines. These RDF stores are a result of several independent research studies and implementations, and most of them have been inspired from proven techniques in database design [19].

A recent study [20] categorizes RDF stores into the following categories:

- The *relational perspective* considers RDF data as just another type of relational data, and reuses the existing techniques developed for storage, retrieval, and indexing of relational database systems.

- The *entity perspective* is resource-centric. It treats resources in an RDF data set as "entities" or "objects", where each entity is determined by a set of attribute-value pairs [21, 22] similar to a classical information retrieval setting.

- The *graph-based perspective* views an RDF data set as a classical graph, where the subject and object parts of each triple correspond to vertices, and the predicates correspond to the directed, labeled edges between them. It aims to support graph navigation and answering of graph-theoretic queries. This perspective originates from research on semi-structured- and graph databases.

There are two types of approaches for storing RDF data in a relational database: Vertically-partitioned approach and Horizontally-partitioned approach. The vertically partitioned store (also referred to as triple store), will create as many two column tables as the number of predicates in the data set. Predicate in a RDF graph shows the relation between a subject and object. Each row in the relation table has a subject and an object satisfying that property. Table 2.2 and Table 2.3 are two sample vertically partitioned tables with properties 'bornIn' and 'graduatedFrom' respectively.

Table 2.2: Sample column table created with predicate 'bornIn'.

Subject	Object
Bill Gates	Seattle
Melinda Gates	Dallas

Table 2.3: Sample column table created with predicate 'graduatedFrom'.

Subject	Object
Bill Gates	LakeSide School
Melinda Gates	Duke University

Query processing on vertically-partitioned tables can be categorized into two types. The first type is when we need to fetch the triples corresponding to a specified property. The second is when the predicate is the unknown value and it needs to be fetched. The first case is just a matter of fetching all the triples corresponding to a relation. A single table scan is enough to achieve this. The second scenario is very expensive, as we have to scan over all the property tables and perform a join operation over them.

One way to store and index the data in a vertical representation is an unclustered index approach, where, data is stored in a single physical triple table and

multiple indexes are created on it. The clustered index approach is a more common approach, where several indexes are created over multiple distinct physical tables. This significantly improves query execution performance, but is inefficient in terms of storage space. Some of the well-known clustered RDF stores are Hexastore [23], Virtuoso [13], TripleT [24], 4store [17] and RDF-3X (discussed in detail later in this section).

In the horizontal partitioned approach, RDF data is stored in a single table that has one column for each predicate value and one row for each subject value. For queries that do not specify a predicate value, the whole table must be analyzed. This will hinder performance significantly when the data set has a large number of predicates. Also, relation schemas are traditionally considered to be static, and change in the schema is not well supported in relational database management system. Therefore, adding a new predicate to the RDF data becomes complicated. On the positive side, integrating existing relational data with a new RDF data set is easy. Jena [14] is an example of a popular horizontally partitioned RDF store.

2.3 Studies comparing RDF stores

Researchers Abadi, Marcus, Madden and Hollenbach in [25] compare the vertically-partitioned approach with the column-oriented approach. In a column-oriented store, tables are stored as a collection of columns instead of rows. Some of their observations were that, by storing data in columns rather than rows, inserts might be slower in column-stores, especially when not done in batch. On the positive side, only the columns relevant to a query are read. Even though both approaches improve the performance and scalability of system, vertical partitioning was observed to be better than column-based approach. A recent study [26] compared four different RDF stores Virtuoso, 4store, BigOWLIM and BigData on LUBM query specifications and observed that, Virtuoso data store was most efficient for loading and querying small amounts of data. It was also observed that the performance of 4store data store was consistent with increase in the data size and query complexity.

The MapReduce framework and its open source implementation Hadoop are now considered to be sufficiently mature, and are widely used across several do-

mains in the industry as well as academia. Recently, researchers have been focusing on the MapReduce family of approaches for developing scalable data processing systems. A recent study [27] concluded that because of the challenges in debugging in a distributed MapReduce model, and because of challenges in dealing with the complex programming model, it is unlikely that MapReduce can substitute database systems.

The key performance determinants for SPARQL query processing using an RDF engine are the indexing methodology employed, the design of the SPARQL query processing engine, support for query optimization, and the query evaluation engine. RDF-3X employs a novel architecture, with innovations in each of these key components. It is shown that RDF-3X outperforms systems such as Monetdb [28] and PostgreSQL [29], and is one of the most efficient RDF data processing engines on a single compute node. Alternate compressed bitmap-based indexing schemes for RDF data include TripleBit [30], BitMat [31], and FastBit-RDF [32]. However, these solutions are not as comprehensive as RDF-3X. In another benchmarking study [33], the NoSQL databases HBase [34], CouchBase [35], and Cassandra [36] have been compared to native RDF stores such as 4store. Results revealed that NoSQL systems are competitive with native RDF stores in terms of query times for simple queries. Complex queries required Map- and Reduce-like operations that introduced a high latency for distributed view maintenance and query execution time. Therefore, RDF stores fared better for complex queries. In this work, we need a very efficient and lightweight single node RDF store to utilize in our distributed architecture. After considering prior work in detail, we chose to extend RDF-3X. It is discussed in more detail in Section 3.

2.4 SPARQL

SPARQL Protocol and RDF Query Language (SPARQL) is a specialized language for querying RDF data sets. SPARQL has been standardized by the RDF Data Access Working Group (DAWG) of the W3C in January 2008, and is now widely supported in RDF implementations. Several triple stores, for example ARC2 [37], BigData [38], RDF-3X have support for the SPARQL query language. SPARQL queries are expressed as triple patterns, and support conjunctions, disjunctions,

and various ways to order the results. Query triple patterns include variables that must be searched for in the RDF data set. The result of the query will be the records that match in the RDF data. To illustrate this, consider the SPARQL query in Figure 2.3.

```
SELECT ?name
WHERE {
  ?person hasName ?name .
  ?person bornIn <http://en.wikipedia.org/wiki/Hodgenville_KY> .
  ?person diedIn <http://en.wikipedia.org/wiki/Washington_D.C.> .
}
```

Figure 2.3: An example SPARQL query.

Variables in a SPARQL query begin with a "?" or "$" (*?name*, *?person*). Variables after the SELECT clause (*?name*) are the results that will be returned. Triples included in the curly braces after the WHERE clause denote the graph pattern that is going to be searched in the RDF graph. Conjunctions are denoted by a dot between the triples.

In order to make queries more readable and shorter, SPARQL allows the definition of prefixes. *prefix:name* is the syntax for prefixed names in a SPARQL query. The prefix in a SPARQL query is allowed to be empty but the value is not. Figure 2.4 is the prefixed version of the query in Figure 2.3. In comparison, the latter is clearly more readable than the former.

```
PREFIX y: <http://en.wikipedia.org/wiki/>
SELECT ?name
WHERE {
  ?person hasName ?name .
  ?person bornIn  y:Hodgenville_KY .
  ?person diedIn  y:Washington_D.C .
}
```

Figure 2.4: A SPARQL query demonstrating the usage of prefixed names.

During the execution of a SPARQL query, a typical processor will go through the following steps:

- Fetch external RDF data defined in the FROM or FROM NAMED clauses,

or directly evaluates the query on the default data set stored in a local RDF store.

- Search for the subgraphs matching the graph pattern in that data.
- Bind the variables of the query to the corresponding parts of matching triple(s).

To demonstrate the execution of SPARQL query, consider the query in Figure 2.3. This query is constructed to return the names of all the persons from the data set in Table 2.1, who were born in Hodgenville_KY and died in Washington_D.C. First, the query processor sees that the user is only interested in the variable *?name*. The variable *?person* is not a part of the projection clause. Nevertheless, variable *?person* is still used for pattern matching and the query processor searches for all the graph triples satisfying this pattern. In this case, it returns `http://en.wikipedia.org/wiki/Abraham_Lincoln` as the result. Note that since query is only returning the name of a person, there may be duplicates in the result. One way to handle this is to use the SPARQL keyword *distinct*. The distinct keyword eliminates the duplicates from the output data. Queries are not always conjunctive and SPARQL has two keywords *UNION* and *OPTIONAL* to handle the disjunctive queries.

2.5 MPI

The Message Passing Interface (MPI) is a language-independent protocol designed to function on a wide variety of parallel computers. Communication between the tasks is achieved either in a point-to-point fashion or by broadcasting messages to a predefined group of tasks. Also, the developer has complete control of program dataflow and execution, thereby supporting data and task parallelism. Having said that, appropriate usage of primitives is critical to obtain correct and optimal results in a MPI program.

MPI provides error handling for runtime errors and profiling tools for monitoring performance, and these are very important for understanding bottlenecks [39]. Furthermore, MPI has well-defined constructors, destructors, and opaque objects, giving it an object-oriented feel. It also supports both the SPMD (Single program,

multiple data) and MPMD (multiple program, multiple data) modes of parallel programming. In addition to this, MPI provides a thread-safe API, which is useful in multithreaded environments as implementations mature and support thread safety themselves.

MPI consists of a set of library routines which can be easily integrated with any standard high level language like Fortran, C, C++, Java or Pascal. In this thesis, we use the C++ API. The MPI acts as the backbone of our implementation, on which the entire distribution mechanism is dependent on. Both RDF3X-MPI*load* and RDF3X-MPI*query* achieve parallelism using MPI on the cluster.

Chapter 3

RDF-3x

RDF-3X [40] is a highly optimized RDF engine capable of handling large-scale RDF data using a RISC-style architecture. RISC is the acronym for Reduced Instruction Set Computing, which is inspired by a CPU design strategy based on the idea that simplified instructions can provide enhanced performance. RDF-3X is a state-of-the-art single-node RDF engine that can perform dynamic query optimization. One drawback of RDF-3X is its scalability with data size. It is limited by memory constraints of the node on which it is executed. Hence, we try to address this issue by introducing parallel data partitioning and MPI. The following sections describe the key features of RDF-3X.

3.1 Storage and Indexing

Several different storage strategies, for instance, relational databases, property tables, etc. have been used in the past to store RDF data. Relational databases create a one-to-one mapping between a subject-predicate-object triple and a row of a database relation. On the other hand, a property table stores triples with a common predicate value [41]. RDF-3X adopts a different and simpler approach by storing the RDF data in a triples table. It maintains a large triples relation with three distinct columns to store the subject-predicate-object of a triple. All triples are internally stored in multiple compressed B+ trees.

Operating with strings directly is not every efficient. Therefore, RDF-3X converts all strings to unique numerical identifiers. This not only makes storage and

compression easier, but also allows for fast RISC-style operators. That said, mapping dictionaries between these numerical values and the string literals need to maintained to support this conversion. Before the start of query execution, string literals are converted into their dictionary integer identifiers. These identifiers are converted back to strings before showing the output to the user. To ensure it can answer any possible query pattern, RDF-3X maintains exhaustive indexes of the subject, predicate and object triples. All six indexes SPO, SOP, OPS, PSO, OSP, POS are stored as compressed B+ trees. Triples in each of these indexes are sorted lexicographically in the order of subject, predicate and object. The sorting results in neighboring tuples to be very similar and hence, compression techniques Gamma and Golomb are applied over these tuples to minimize redundant information storage.

To take care of the scenario where the query pattern has two unknown variables, RDF-3X maintains six additional aggregate indexes SP, PS, SO, OS, PO, OP. These indexes are also maintained in a B+ tree, whose leaves contains the two values of the unknown variable and a count. This count represents the number of times the pair occurs in the entire set of triples. Finally, RDF-3X has three more fully aggregated indexes S, P, O stored in B+ trees containing just (value, count) entries. Even though there is redundancy in the indexes created, the compression schemes used make this approach feasible. Furthermore, accesses to the indexes reduces the cost of query evaluation.

3.2 Query Processing

The first step of processing a SPARQL query is constructing a query graph. This representation helps us match the query triple pattern with the actual RDF graph. The query string is translated into a set of triple patterns, each containing variables and constants. The constants in the triple patterns are replaced by their ids in the mapping dictionary. If the query consists of a single triple pattern, then a direct index scan of the one of the different indexes created will give us the result. If query has multiple triple patterns, a join operation has to be performed on the results from individual patterns.

In the query graph, each node corresponds to one query pattern. This query graph forms the basis of developing a query plan in the later stages. A naive execution plan can be constructed from a query graph by translating nodes into range scans, edges into joins. RDF-3X also supports disjunctive queries by allowing the use of SPARQL keywords like UNION and OPTIONAL. Next, apply specific operations on the output data according to the option specified like FILTER or DISTINCT. Finally, the mapping dictionary is used to convert the integer ids back to string literals.

3.3 Query Optimization

Join ordering is one of the key issues in making RDF-3X query processing optimal. In past work, dynamic programming-based approaches [42] have been explored to come up with an optimal join ordering. RDF-3X opts for a different approach using bottom-up dynamic programming framework [43] to generate a near optimal plan. Bottom-up dynamic-programming maintains optimal plan(s) for each subgraph in a dynamic programming table. It is not feasible to compute the plan quickly in cases where the SPARQL queries have large number of joins. Bottom-up approach would be the ideal choice for such scenarios, because it provides fast enumeration compared to the top-down approach.

The seeding in this Bottom-up approach is a two step process. The optimizer initially checks for the query variables that are bound in other parts of the query and if the variable is unused, it uses aggregated index for projection. In the subsequent step, the optimizer decides which of the applicable indexes to use. If there are constants in the triple pattern, it will form a prefix for one of the possible indexes, allowing for range scans in that index. Other indexes might produce results with an order suitable for a subsequent merge-join. This results in the generation of multiple plans and the optimizer keeps track of all the costs of the generated plans. The ones with lower overall cost are preferred and the rest are pruned.

3.4 Selectivity Estimation

To predict the costs of different execution plans for a SPARQL query, RDF-3X requires selectivity estimators for selections and joins. RDF-3X uses two kinds of statistics to achieve optimal results. First and the generic approach is to use specialized histograms. Second uses frequent join paths in the data and gives more accurate predictions on these paths for large joins.

Histogram approach is simpler out of the two and its methodology is to build histograms on the combined values of SPO triples. Using this, the exact number of matching triples can be gathered with one index lookup for each literal or literal pair. This is not sufficient for estimating join selectivities. Also, it is required that histograms are always read from main memory. That said, the volume of RDF data sets has increased to the billion triples range. Hence, trade-off has to be made between the accuracy of the estimation histograms and the space allocated to them in the main memory.

To tackle such problems with large data sets, a different approach has been employed. This approach leverages all SPO permutations, binary, and unary projections with counts that RDF-3X has already generated. In this approach, the join selectivity of a triple pattern is defined as the degree to which this pattern is selective with other triple patterns. The results of the join selectivities are stored in compressed B+-trees indexed by the constants in the triple pattern.

In summary, RDF-3X is a very optimal single node RDF data processing engine that outperforms its peers. The index structure, compression scheme, and query optimizer all play a pivotal role in making RDF-3X a very fast SPARQL query processing engine. Hence, we chose to use RDF-3X in this work.

Chapter 4

Data Partitioning

In this chapter, we give a detailed description of our data partitioning and indexing architecture. The main intention of our distributed architecture is to split the large data set into partitions that can fit in-memory. We achieve this using a parallel hash partitioning algorithm. Since all the partitions completely fit in the memory, we don't need to request any secondary storage for the data. Thereby, there is no compromise in the run time of the store even if the system is distributed. Also, we perform operations like n-hop guarantee which ensure that the partitions are independent of each other. This will avoid the need for any communication between the parallel tasks. Each of these independent tasks are provided with state-of-the-art RDF3X*load* module that loads and creates a database of its own.

The MPI data partitioning utility RDF3X-MPI*load* consists of four main steps

- Preprocess the RDF data set to create integer triples and literal to integer ID map.
- Assigning vertices to MPI nodes using a parallel Hash partitioning algorithm.
- Generating triples in each partition using n-hop guarantee.
- Using RDF3X*load* to create the partition databases.

To facilitate the hash partitioning, we create unique integer ID's for every literal using a Hash table. Using these ID's, we generate the corresponding integer triples from the original RDF data set, which servers as the input graph from this

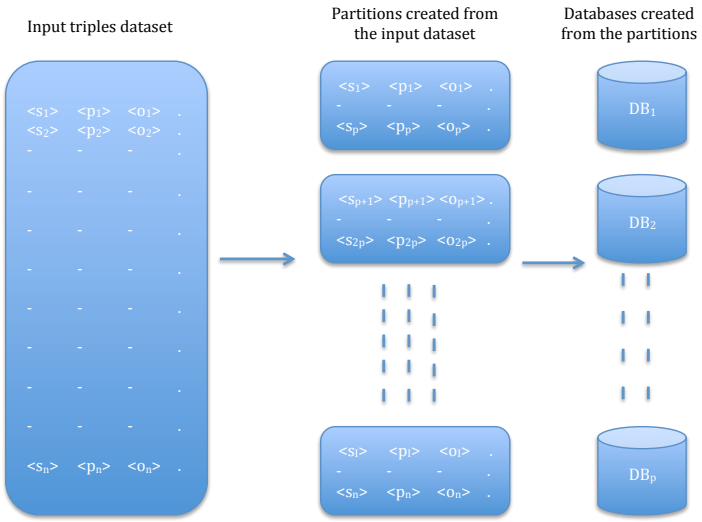


Figure 4.1: Overview of RDF3X-MPI: We introduce a new distributed data partitioning utility RDF3X-MPI*load* and RDF3X-MPI*query*, a distributed version of the RDF-3X utility RDF3X*query*.

step. This is not only efficient in terms of memory, but also provides a simple way to store the graph in the form of arrays. Arrays in turn result in a very fast storage and retrieval of information from the RDF graph.

The next step is random assignment of vertices to the partitions using a hash-based parallel vertex partitioning algorithm. The simplicity and distributed nature of this algorithm makes it very effective even for large RDF data sets. Number of partitions created by this approach is equal to the number of MPI processors set. Since, the vertices are randomly assigned, the hash-based technique achieves balanced partitions and is very fast (shown in Figure 6.3). The vertices assigned to a partition are called the *Core* vertices [44]. The core vertex of one partition is unique across all the partitions. Therefore, we use them to avoid duplicates while printing the results, by only allowing the triples whose source node belongs to the core vertex set.

4.1 Partitioning Algorithm

Our parallel partitioning algorithm uses a simple Hash function to assign vertices to every node. Each node t_{id} is assigned all the vertices V_i that satisfy

$T_{id} = V_i \bmod(p)$, where p is the number of partitions.

All the vertices assigned to a partition are the Core vertices. A naive triple placement strategy is to include all the triples whose subject is a core vertex of the partition. This method is very efficient for star shaped queries but performs poorly for complex queries [45, 46]. To illustrate this, consider the following SPARQL query corresponding to the RDF graph in Figure 2.2:

```
SELECT ?name
WHERE {
  ?person hasName ?name .
  ?person title ‘‘President’’ .
  ?person gender ‘‘Male’’ .
  ?person diedIn ‘‘1865-04-15’’ .
}
```

Figure 4.2: Sample star shaped SPARQL query.

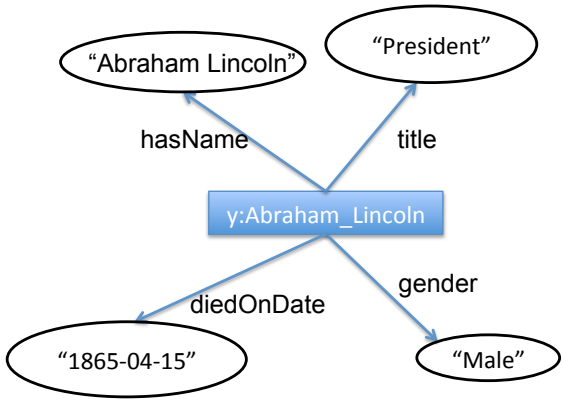


Figure 4.3: Query graph corresponding to Figure 4.2.

Figure 4.3 shows the star shaped query graph corresponding to the above query. Since, all triples corresponding to a subject belong to one of the partitions (based on our triple placement strategy). The partition containing the subject

Y:Abraham_Lincoln will return the correct result and rest of the partitions will not print any output. Now, consider a complex query corresponding to the RDF graph in Figure 2.2:

```
SELECT ?country
WHERE {
  ?country hasName ‘‘United States’’ .
  ?country foundYear ‘‘1776’’ .
  y:Hyde_Park_NY locatedIn ?country .
  y:Hyde_Park_NY foundingYear ‘‘1718’’ .
}
```

Figure 4.4: Sample complex SPARQL query.

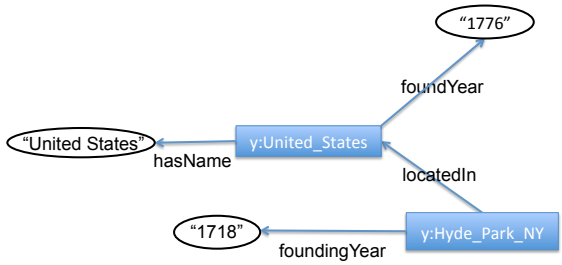


Figure 4.5: Query graph corresponding to Figure 4.4.

Figure 4.5 shows the query graph corresponding to the above query. From the query graph we can clearly see that none of the partitions can execute the query completely on its own. Partial results from different partitions need to be joined in order to obtain the final output. But, inter-processor communication is a highly time consuming step, especially if the intermediate results are large. Therefore, we try to avoid this as much as possible. In order to achieve this, we perform n-hop guarantee on every node.

4.2 n-hop guarantee

Undirected n-hop guarantee [44] ensures that every edge that can be visited within n hops from the current vertex is included in the partition. The implementation of an undirected n-hop guarantee is very similar to that of Breadth First Search

(BFS) on a graph. Let vertex set V denote the vertices assigned to a partition (also known as Core vertices), after the hash partitioning step. Mark all the vertices of V in a vertex-bitmap *map*. Undirected 1-hop triples include all the edges whose source or destination is marked in the *map*. For undirected 2-hop triples, we first gather all the vertices that are at a distance of 1 hop from every vertex marked in V into a new set *V'*. Next, we iterate through all the edges in the graph and include ones whose source or destination vertex belongs to *V'*. This gives us the undirected 2-hop generated triples. Similarly for n-hop, we implement 1-hop guarantee on the vertices generated at the (n-1) level and generate new set of triples corresponding to the bitmap *map*. Algorithm 1 describes a more formal implementation of the n-hop guarantee.

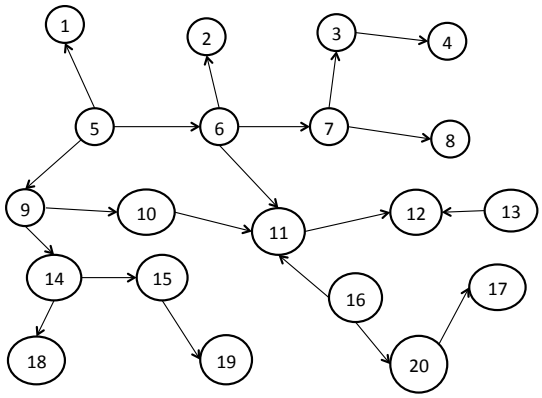


Figure 4.6: Sample directed graph with vertices numbered from 1 to 20.

To illustrate this, consider the following graph in Figure 4.6. This is directed graph with vertices numbered from 1 to 20. Undirected 2-hop implementation on the vertex *11* includes the following two steps. First, we first gather all the vertices that can be visited within one hop and mark them in a vertex-bitmap. This set includes the vertices *6, 7, 12, 16, 10*. Second, we iterate through all the edges in the graph and include the ones whose source or destination is in the bitmap. Edges marked in red indicates the triples included by the undirected 2-hop guarantee on the vertex *11*.

As the final step of the data partitioning methodology, each MPI task creates its corresponding database by calling the RDF3X*load* library on its partition.

Algorithm 1 Undirected n-hop guarantee

```
bool vertexBitMap[] <– Initialize vertex bitmap array
startVertex = vertexBlock[partId-1];
endVertex = vertexBlock[partId];
hopCount = 2;                          ▷ Initialized depending on the parameter;

▷ Array vertexBlock contains the range of vertices for a partition Id
▷ Boolean bitmap vertexBitMap marks n-hop vertices
▷ Vertices from startVertex to endVertex are the Core vertices of this partition
for currVertex = startVertex to endVertex do
    nHopHelper(currVertex, adjacencyArray, offsetArray, hopCount-1)
end for

for index = 0 to inputTripleCount do        ▷ Iterating through all the triples
    if vertexBitMap [sourceVertex [index]] ||vertexBitMap [destinationVertex [index]] then
        addEdgeToPartition(source[index], desination [index], edge [index]);
    end if
end for

▷ adjacencyArray is an array representation of the adjacency list
▷ offsetArray is an integer array that points to the start of a list of neighboring edges
function nHopHelper(currVertex, adjacencyArray[], offsetArray[], hopCount)
    vertexBitMap[currVertex] = true;
    if (hopCount == 0) then
        return;
    end if
    offsetStart = 0;
    if (currVertex != 0) then
        offsetStart = offsetArr[currVertex-1];
    end if
    offsetend = offsetArr[currVertex];
    for index = offsetStart to offsetEnd do
        nHopHelper(adjacencyArray[index], adjacencyArray, offsetArray, hopCount-1)
    end for
end function
```

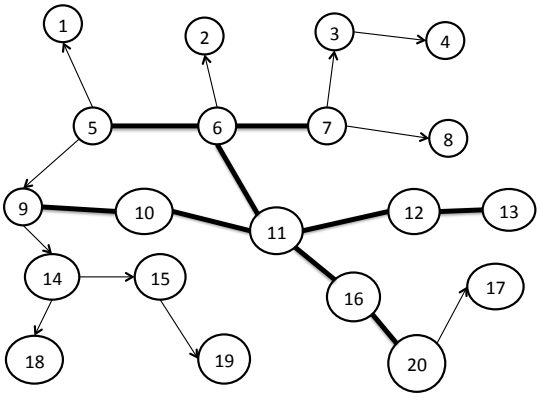


Figure 4.7: Undirected 2-hop guarantee implementation on the vertex *11* of graph in Figure 4.6. Edges included are shown in bold lines.

Chapter 5

Query Processing

RDF3X-MPI*query* is a parallel query processing module that processes SPARQL queries. It uses MPI to process queries in a distributed environment. It has two main steps: First, loading databases in parallel using MPI tasks. Second, executing the query independently on each partition, in parallel using RDF3X*query* utility.

Initially, we create MPI tasks, same as the the number of databases generated using RDF3X-MPI*load*. Each task uses RDF3X*query* to load the corresponding database and waits for the input query. When the root receives a input SPARQL query, it broadcasts the query to all other tasks. Each task then runs the query on its database asynchronous to other tasks to generate the partial results. Since, n-hop guarantee ensures that the partitions are independent of each other, we don't need to perform extra operations on the partial results. We avoid duplicates in the partial results using the *Core* vertexes assigned to each partition. Unlike a single node RDF store, RDF3X-MPI runs each task on a separate node. Thereby, it is highly scalable as the resources are only bound by the number of nodes in the cluster.

Distance of the farthest edge (DoFE) [44], is the distance from a vertex to the farthest edge in the graph. Here, distance between two vertexes is defined as the number of undirected hops between them. Formal definition is shown in Algorithm 2.

Algorithm 2 Distance of the Farthest Edge - DoFE.

```
procedure DoFE(Graph G, Vertex v)
    maxDist = 0;
    for Edge e(v1,v2) in G do
        dist = min(Distance(v, v1), Distance(v, v2)) + 1;
        if dist > maxDist then
            maxDist = dist;
        end if
    end for
    return maxDist;
end procedure
```

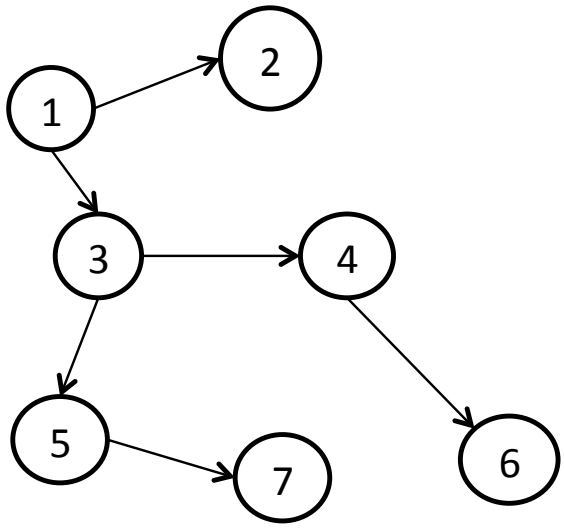


Figure 5.1: Example graph to illustrate Distance of the farthest edge concept.

To illustrate this further, consider a small graph in Figure 5.1. For vertex *1*, DoFE is calculated by finding the farthest edge from it. Both edges (*5,7*) and (*4,6*) are at a distance of 3 from vertex *1* and rest of the edges have distance less than 3. Hence, DoFE(*1*) = 3.

Center or *Core* [44] of a query graph is defined as the vertex which has the least DoFE value. Consider the graph in Figure 5.1 to illustrate this. Vertex *3* has the least DoFE value of 2, compared to all other vertices, as it can reach any other vertex within two undirected hops. Hence, *3* is the Center of the graph. In the following sections of this thesis, if we refer to DoFE of a graph, it is implied that

the value corresponds to the DoFE from the center of the graph. Formal procedure to find the *Center* is shown in Algorithm 3.

Algorithm 3 Find Center of a Query graph.

```
procedure GetCenter(Graph G)
    minDofe = INTEGER_MAXVALUE;
    center = RANDOM_VERTEX                    ▷ Initialized to any vertex
    for Vertex v in G.getVertexes() do
        dofe = DoFE(G, c);
        if dofe < minDofe then
            minDofe = dofe;
            center = v;
        end if
    end for
    return v;
end procedure
```

Every query in LUBM has its DoFE value less than or equal to 2. Thereby, every query in LUBM is Parallelizable without communication (PWoC) [44] on an undirected 2-hop implemented partition set. Some RDF data sets like LUBM, have vertexes with very high indegree ($\times 10^6$). Undirected 2-hop guarantee on such vertexes will result in massive replication of triples across all partitions as shown in Figure 6.2. Such high replication results in an increase in the size of individual partitions, which in-turn effects the query run times. Therefore, it is very inefficient to execute a query on an undirected 2-hop implemented partition, which only requires a 1-hop forward implementation, as the size of the former is significantly larger compared to the latter [46]. Therefore, a logical way to approach this problem is to create several partition variations beforehand and use a partition set corresponding to the DoFE value of the input query. Therefore, we create the following variations for our experiments with LUBM and DBPSB data sets.

- 1-hop-f(forward): Includes all the outgoing triples from a vertex
- 1-hop-u(undirected): Includes both incoming and outgoing triples from a vertex

- 2-hop-f(forward): Includes all the outgoing triples from a vertex, recursively for two hops
- 2-hop-u(undirected): Include both incoming and outgoing triples from a vertex, recursively for two hops

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X, ?Y1, ?Y2, ?Y3
WHERE {
  ?X rdf:type ub:Professor .
  ?X ub:worksFor <http://www.Department0.University0.edu> .
  ?X ub:name ?Y1 .
  ?X ub:emailAddress ?Y2 .
  ?X ub:telephone ?Y3 .
}
```

Figure 5.2: LUBM query 4.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X, ?Y, ?Z
WHERE {
  ?X rdf:type ub:GraduateStudent .
  ?Y rdf:type ub:University .
  ?Z rdf:type ub:Department .
  ?X ub:memberOf ?Z .
  ?Z ub:subOrganizationOf ?Y .
  ?X ub:undergraduateDegreeFrom ?Y .
}
```

Figure 5.3: LUBM query 2.

Consider query 4 from the LUBM data set as shown in Figure5.2. textit?X is the center of the query graph as the DoFE is the least from it. Implementing a 1-hop forward guarantee on $?X$ includes all the outgoing edges from $?X$ and thereby the complete query graph. Consequently, a 1-out guarantee on every partition would include all possible sub-graphs that match this pattern. Hence, such queries with DoFE-1 can be evaluated without the need for any inter-node communication

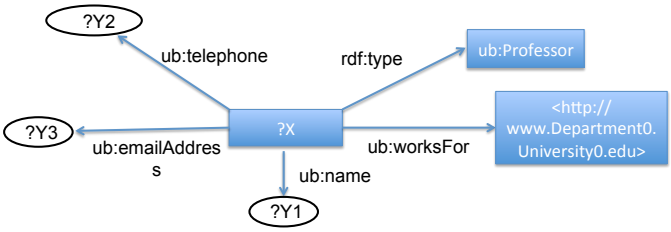


Figure 5.4: Query graph of query in Figure 5.2.

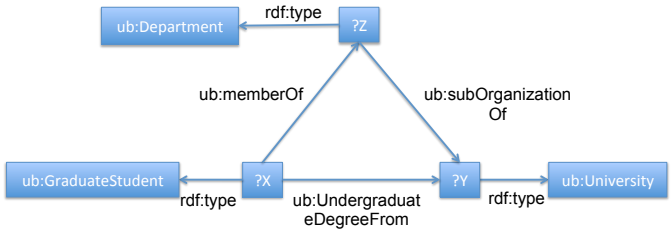


Figure 5.5: Query graph for query in Figure 5.3.

after 1-hop guarantee.

For LUBM query 2 in Figure 5.5, all three vertexes *?X*, *?Y* and *?Z* all have the least DoFE value of 2. That said, *?X* only requires a 2-hop forward guarantee to include all the edges in the graph. Therefore, *?X* is the center of the query graph. Implementing a 2-hop forward guarantee on every Core vertex will include all possible sub-graphs that match this pattern. Consequently, the partitions can be executed independently without the need for any inter-node communication.

A SPARQL query can be categorized into two types. First, query that requires inter task communication. Second, query that doesn't require inter task communication. From the above examples, it is clear that, as long as DoFE(query) <= n (from n-hop guarantee), the query can be independently executed on all the partitions in parallel. Since, we have multiple partition variations generated beforehand, we can accommodate many different types of queries. Having said that, it is impossible to predict the complexity of a query in a real-time environment. If DoFE(query) > n, we would have to evaluate parallelizable sub-queries on different nodes and perform joins on the intermediate results to get the final output. Handling such queries is not a part of this thesis and is a work in progress.

Chapter 6

Results and Obersvations

In this section, we report the performance analysis of RDF-3X and RDF3X-MPI mainly on LUBM and DBPSB data sets.

6.1 Benchmarks used

There are now several popular software packages [47] for benchmarking RDF stores for their SPARQL query processing capabilities. BillionTriples [48], Uniprot [4], Yago [2] are some of the commonly used RDF data sets. We use four RDF data sets for our experiments 6.1. From the famous RDF benchmark: LUBM (Leigh University Benchmarks) [49], we generate LUBM100(14M) and LUBM1000(138M) representing data sets for 100 and 1000 universities respectively. From DBPSB (Dbpedia SPARQL Benchmark) [3], we use a data set DBPSB(279M from `http://dbpedia.aksw.org/benchmark.dbpedia.org/`) and finally, from BSBM (Berlin SPARQL Benchmark) [50] we generate a data set BSBM(71M). For the experiments, we use all 14 queries from LUBM benchmark and 9 queries from queries from DBPSB shown in the Appendix **??**.

For experiments, we use a cluster with 32 nodes. Each compute node is a Dell PowerEdge R610 server with 2 quad-core Intel Nehalem processors running at 2.66 GHz, 24 GB of RAM and 1 TB SATA drives at 7200 RPM. Six HP ProCurve 2910 network switches connect all compute nodes at 1 GbE speeds. All six switches are interconnected with 10 GbE and the master switch has a 10 GbE uplink to core storage and infrastructure servers.

Table 6.1: RDF benchmark data used in this study.

RDF data set	# triples ($\times 10^6$)	$\lvert SO \rvert$ ($\times 10^6$)	$\lvert P \rvert$
LUBM-100	14	3.3	18
LUBM-1000	138	33	18
BSBM	71	15	29
DBPSB	279	84	39675

6.2 Split-up query execution time

Table 6.2 shows the step by step analysis of the query processing utility of RDF3x on the data sets LUBM-100 and LUBM-1000. Columns 2 - *Query Parsing*, indicates the time taken to parse the input query and create a query graph. Column 3 -*Running Query Optimizer* indicates the time taken to generate a plan to evaluate the query. Column 4 - *String Lookup* shows the time taken by RDF3x to convert the integer ID's to literals before printing them. We don't report the other steps like *Evaluating query tree* and *Collecting output values* as they often take less than a millisecond to run.

All the queries except 2,6,9 and 14 have a very optimal run-time. The string lookup step is drastically effecting the run-time for these queries. The reason for this behavior is that the output size of the queries 2,6,9 and 14 is significantly large compared to the other queries. RDF-3X iterates over a STL MAP of size same as that of the output. These queries have output size in millions, resulting in iterating over a huge STL MAP. This explains to an extent, the very high string lookup times for these queries, as iterating though an STL MAP is an inherently slow operation.

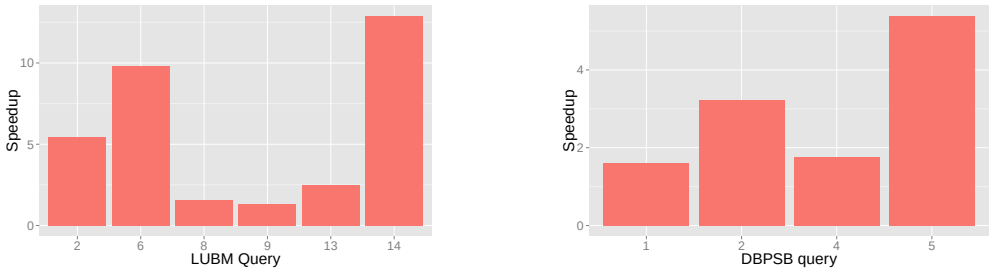
6.3 RDF3X-MPI query time analysis

Table 6.3, shows the query run-time for our RDF3X-MPI on LUBM-1000. We report the query evaluation time for several partition variations upto 32. Figure 6.1 shows that queries with large output in both LUBM and DBPSB achieve significant speedup compared to RDF-3X. For instance, queries 2, 6 and 14 achieve 4.4x, 3.7x

Table 6.2: Split-up of RDF-3X query execution time (ms) on LUBM data.

LUBM-100				
Query	Q. Parse	Q. Opt	Str. lookup	Total time
1	171	156	38	508
2	66	44	9359	10306
3	70	18	0	96
4	126	18	0	198
5	31	14	1	86
6	0	0	41842	42631
7	34	74	0	166
8	9	2	124	563
9	42	2	6306	7487
10	0	1	0	1
11	19	2	0	40
12	1	1	0	48
13	0	15	1	188
14	0	0	823	1388
LUBM-1000				
Query	Q. Parse	Q. Opt	Str. lookup	Total time
1	149	242	29	529
2	52	52	84252	90272
3	61	42	14	148
4	191	30	0	351
5	50	21	49	187
6	0	0	419728	427617
7	62	157	0	289
8	13	1	42	2838
9	39	2	91830	103465
10	0	1	0	1
11	7	1	24	93
12	20	1	0	47
13	0	33	130	970
14	0	0	7331	13533

and 16x speedup (for 16-way partitioning) and 5.4x, 9.8 and 12.9x (for 32-way partitioning). Similarly, on DBPSB data set using 16-way MPI-task parallelism, queries 2, 5 achieved a speedup of 3.2x, 5.4x respectively. For queries with smaller output size in both LUBM and DBPSB, speedup observed is negligible because,



(a) LUBM Speeup on 32 nodes.

(b) DBPSB Speedup on 16 nodes.

Figure 6.1: Speedup on RDF3X-MPI over RDF-3X. Slowdowns are not indicated.

the overhead of MPI task creation is more than the speedup achieved using the parallelism.

Table 6.3: LUBM-1000 query run-time on RDF3X-MPI.

		Num. Partitions				
Query	**DoFE**	2	4	8	16	32
1	1-out	504	702	791	599	544
2	2-out	144289	82423	71186	20347	16535
3	1-out	271	343	159	375	437
4	1-out	407	486	347	788	849
5	1-out	221	216	168	260	447
6	1-out	249754	240752	120633	114172	43665
8	2-out	6167	3897	8889	3634	1786
9	2-out	282021	187415	236520	92249	76991
10	1-out	1	5	2	3	68
11	2-out	74	117	167	98	134
12	2-out	197	135	161	140	571
13	1-out	4826	4615	1508	2400	392
14	1-out	6126	9175	4607	802	1048

6.4 Load imbalance factor

It is very critical for the work to be balanced across every task in an MPI job. One lagging task can degrade the run-time of the whole process. Table 6.5, shows the load imbalance factor of RDF3x-MPI on LUBM1000. It is the ratio of time taken by the slowest task and average task time. Higher value of load imbalance

Table 6.4: DBPSB Query run-time (ms) on RDF-3X and RDF3X-MPI using 16-way partitioning.

		Query time	
Query	DoFE	RDF3x	RDF3x-MPI
1	1-out	501	310
2	1-out	1084523	336885
3	1-out	592	1323
4	1-out	716	409
5	1-out	553210	102659
6	1-out	155	182
7	1-out	147	242
8	1-out	135	208
9	1-out	123	150

factor, primarily indicates the load imbalance among the partitions. For most of the cases, the value is less than 2 in the Table 6.5, which indicates the partitions are fairly balanced. This behavior confirms that assignment of vertices using a hash-partitioning methodology provides good load balance.

Table 6.5: Load imbalance factor of RDF3X-MPI on LUBM query run-time.

	Partitions		
Query	2	16	32
1	1.0	2.1	2.2
2	1.0	1.1	1.3
3	1.2	2.1	2.7
4	1.0	1.6	1.5
5	1.1	1.9	2.0
6	1.0	1.7	1.3
8	1.0	2.3	3.3
9	1.9	1.1	1.1
11	1.1	1.5	2.1
12	1.4	1.5	5.5
13	1.0	1.7	1.4
14	1.0	1.1	2.5

6.5 Replication ratio

We define *Replication Ratio* as the number of times a triple is replicated across all the partitions or formally, the ratio of total number of triples created in all the partitions and number of triples in the input data set. Therefore, after applying n-hop guarantee, higher replication ratio indicates that the graph is tightly connected. Most of the queries in LUBM just require a 1-out or 2-out guarantees to ensure complete PWoC. Some real-time queries might be more complicated and may require undirected 2 hop or even larger hop guarantees to ensure PWoC across all tasks. Such larger hop guarantees will result in massive replication of triples across the partitions, negating the speedup achieved by the MPI parallelism.

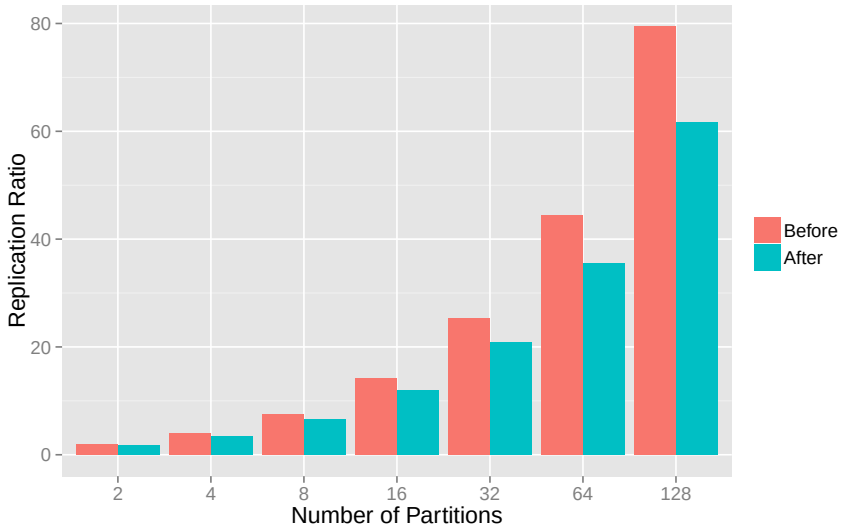


Figure 6.2: LUBM-1000 Replication Ratio before and after high degree vertex removal, upto 32-way partitioning.

Figure 6.2 shows the replication ratio of LUBM-1000 after undirected 2 hop guarantee implementation. Careful analysis of the degree distribution graphs of LUBM-1000 shows that the some vertices have very high indegree, that are resulting in the high replication of triples after n-hop guarantee. Mainly, two vertices have an indegree close to a million (about 7% of the complete graph). End points of these million triples can be spread across several partitions and undirected 2-hop guarantee on all these vertices would result in the replication of same million triples across all the partitions. To overcome this, we propose a solution to separate

these high degree vertices into a new partition at the vertex partitioning phase. This separation is also maintained while implementing the n-hop guarantee by ignoring any high degree vertex encountered from a normal partition. It is clearly evident from the figure that, there is a significant decrease in the replication ratio value after removing the high degree vertices, especially for 64-way and 128-way partitioning.

6.6 Load times

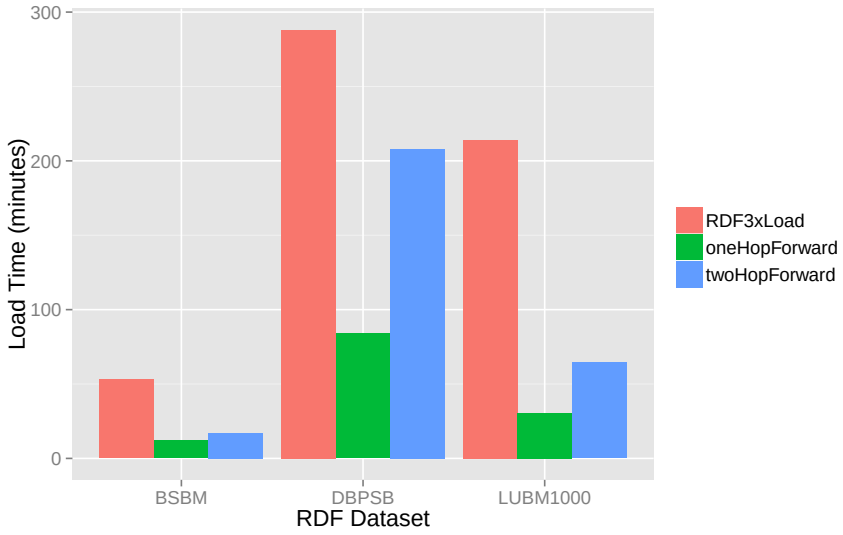


Figure 6.3: Load Time comparison between and RDF3X*load* and two partition variations of RDF3X-MPI*load* using 16-way partitioning.

Figure 6.3 compares the load time of two partition variations of RDF3X-MPI*load* with RDF3X*load*. We use all three data sets LUBM-1000, BSBM, DBPSB and report the load time for 16-way MPI partitioning. 1-hop forward and 2-hop forward are the hop guarantee variations used on the partitions. The load times shown here also include the preprocessing time for creating the integer triples file. In every case, RDF3X-MPI outperforms RDF-3X and the speedup observed is upto 7x.

Chapter 7

Conclusion and Future Work

In this thesis, we propose a distributed RDF data processing engine RDF3X-MPI. Our key contributions include two modules RDF3X-MPI*load* and RDF3X-MPI*query*. Both the modules use MPI library for efficient execution on distributed-memory systems. RDF3X-MPI*load* is the parallel storage and indexing utility that internally utilizes RDF3X*load* to create the database files. First, it creates several vertex partitions using a parallel hash partitioning algorithm. Subsequently, each partition implements n-hop guarantee for triple placement. Finally, each partition creates its own database using the RDF3X*load* utility. RDF3X-MPI*query* is a MPI based distributed query processing module. Each task is equipped with a RDF3X*query* utility that loads the databases in parallel. A query is executed on every task in parallel and all the partial outputs are aggregated to get the final result. We also evaluate performance of both these modules using LUBM, DBPSB and BSBM on a cluster system. With respect to load times, RDF3X-MPI outperforms RDF-3X on every data set and the speedup observed is up to 7×. For query processing time, we observe that the speedup is proportional to the size of the output generated by a query. This distributed system approach can support several terabytes of main memory. Handling large data sets is not the only advantage with this approach, we have also observed significant speedup, especially for queries with large output.

Our work can be extended in the following areas. First, in the hash partitioning technique, the vertices are not assigned to a node based on their locality. Therefore, adjacent nodes might be assigned to different partitions. Instead, if the partitioning

was based on locality, the n-hop guarantee on adjacent vertices would result in overlap of edges resulting in lesser replication of triples. Next, for complex queries with large DoFE values, that are not independently executable on any of the partition variations, we need to break it into parallelizable sub-queries and use MPI function calls to communicate the intermediate results and perform additional join operations to compute the output.

Bibliography

[1] (2004), "RDF Primer. W3C Recommendation." http://www.w3.org/TR/rdf-primer, [Last accessed July 2014].

[2] Suchanek, F. M., G. Kasneci, and G. Weikum (2007) "Yago: A Core of Semantic Knowledge," in *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pp. 697–706.
URL http://doi.acm.org/10.1145/1242572.1242667

[3] Morsey, M., J. Lehmann, S. Auer, and A.-C. N. Ngomo (2011) "DBpedia SPARQL Benchmark - Performance Assessment with Real Queries on Real Data," in Aroyo et al. [8], pp. 454–469.

[4] "Uniprot RDF," http://dev.isb-sib.ch/projects/uniprot-rdf/, [Last accessed July 2014].

[5] PrudâĂŹhommeaux, E. and A. Seaborne (2008), "SPARQL Query Language for RDF. W3C Working Draft 4," http://www.w3.org/TR/rdf-sparql-query/, [Last accessed July 2014].

[6] Forum, T. M. (2009), "MPI: A Message Passing Interface," http://www.mpi-forum.org/, [Last accessed July 2014].

[7] Alani, H., L. Kagal, A. Fokoue, P. T. Groth, C. Biemann, J. X. Parreira, L. Aroyo, N. F. Noy, C. Welty, and K. Janowicz (eds.) (2013) *The Semantic Web, Part II*, vol. 8219 of *Lecture Notes in Computer Science*.

[8] Aroyo, L., C. Welty, H. Alani, J. Taylor, A. Bernstein, L. Kagal, N. F. Noy, and E. Blomqvist (eds.) (2011) *The Semantic Web, Part I*, vol. 7031 of *Lecture Notes in Computer Science*.

[9] Gutierrez, C. (2011) "Modeling the Web of Data (Introductory Overview)," in *Reasoning Web. Semantic Technologies for the Web of Data* (A. Polleres,

C. dâĂŹAmato, M. Arenas, S. Handschuh, P. Kroner, S. Ossowski, and P. Patel-Schneider, eds.), vol. 6848 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 416–444.
URL http://dx.doi.org/10.1007/978-3-642-23032-5_8

[10] Brickley, D. and R. V. Guha (2004), "RDF vocabulary description language 1.0: RDF Schema," http://www.w3.org/TR/rdf-schema/, [Last accessed July 2014].

[11] Group, W. W. (2009), "OWL 2 web ontology language," http://www.w3.org/TR/owl2-overview/, [Last accessed July 2014].

[12] W3C (2010), "URI," http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/#section-URIspaces, [Last accessed July 2014].

[13] Erling, O. (2008) "Towards Web Scale RDF," in *Scalable Semantic Web Knowledge Base Systems - SSWS*.

[14] Owens, A. (2008), "Clustered TDB: A Clustered Triple Store for Jena," [Last accessed July 2014].

[15] Broekstra, J., A. Kampman, and F. v. Harmelen (2002) "Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema," in *Proceedings of the First International Semantic Web Conference on The Semantic Web*, ISWC '02, Springer-Verlag, pp. 54–68.
URL http://dl.acm.org/citation.cfm?id=646996.711426

[16] "Stardog RDF database," http://stardog.com, [Last accessed July 2014].

[17] Harris, S., N. Lamb, and N. Shadbolt (2009) "N.: 4store: The Design and Implementation of a Clustered RDF Store," in *Scalable Semantic Web Knowledge Base Systems - SSWS*, pp. 94–109.

[18] "Franz inc. allegrograph 4.2 introduction," http://www.franz.com/agraph/support/documentation/v4/agraph-introduction.html, [Last accessed July 2014].

[19] Sakr, S. and G. Al-Naymat (2010) "Relational Processing of RDF Queries: A Survey," *SIGMOD Rec.*, **38**(4), pp. 23–28.
URL http://doi.acm.org/10.1145/1815948.1815953

[20] Luo, Y., F. Picalausa, G. Fletcher, J. Hidders, and S. Vansummeren (2012) "Storing and Indexing Massive RDF Datasets," in *Semantic Search over the Web* (R. De Virgilio, F. Guerra, and Y. Velegrakis, eds.),

Data-Centric Systems and Applications, Springer Berlin Heidelberg, pp. 31–60.
URL http://dx.doi.org/10.1007/978-3-642-25008-8_2

[21] Weikum, G. and M. Theobald (2010) "From Information to Knowledge: Harvesting Entities and Relationships from Web Sources," in *Proceedings of the Twenty-ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS, pp. 65–76.
URL http://doi.acm.org/10.1145/1807085.1807097

[22] Fletcher, G. H. L., J. Van Den Bussche, D. Van Gucht, and S. Vansummeren (2010) "Towards a Theory of Search Queries," *ACM Trans. Database Syst.*, **35**(4), pp. 28:1–28:33.
URL http://doi.acm.org/10.1145/1862919.1862925

[23] Weiss, C., P. Karras, and A. Bernstein (2008) "Hexastore: Sextuple Indexing for Semantic Web Data Management," *Proc. VLDB Endow.*, **1**(1), pp. 1008–1019.
URL http://dx.doi.org/10.14778/1453856.1453965

[24] Fletcher, G. H. and P. W. Beck (2009) "Scalable Indexing of RDF Graphs for Efficient Join Processing," in *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09, pp. 1513–1516.
URL http://doi.acm.org/10.1145/1645953.1646159

[25] Abadi, D. J., A. Marcus, S. R. Madden, and K. Hollenbach (2007) "Scalable Semantic Web Data Management Using Vertical Partitioning," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB, pp. 411–422.
URL http://dl.acm.org/citation.cfm?id=1325851.1325900

[26] Patchigolla, V. (2011), "Comparison of Clustered RDF Data Stores," http://www.systap.com/, [Last accessed July 2014].

[27] Sakr, S., A. Liu, and A. G. Fayoumi (2013) "The family of mapreduce and large-scale data processing systems," *ACM Comput. Surv.*, **46**(1), p. 11.

[28] "Monetdb," http://monetdb.cwi.nl/, [Last accessed July 2014].

[29] Neumann, T. and G. Weikum (2008) "RDF-3X: A RISC-style Engine for RDF," *Proc. VLDB Endow.*, **1**(1), pp. 647–659.
URL http://dx.doi.org/10.14778/1453856.1453927

[30] Yuan, P., P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu (2013) "TripleBit: a Fast and Compact System for Large Scale RDF Data," *PVLDB*, **6**(7), pp. 517–528.

[31] Atre, M., V. Chaoji, M. J. Zaki, and J. A. Hendler (2010) "Matrix "Bit" Loaded: A Scalable Lightweight Join Query Processor for RDF Data," in *Proceedings of the 19th International Conference on World Wide Web*, WWW, pp. 41–50.
URL http://doi.acm.org/10.1145/1772690.1772696

[32] Madduri, K. and K. Wu (2011) "Massive-Scale RDF Processing Using Compressed Bitmap Indexes," in Cushing et al. [51], pp. 470–479.

[33] Cudré-Mauroux, P., I. Enchev, S. Fundatureanu, P. T. Groth, A. Haque, A. Harth, F. L. Keppmann, D. P. Miranker, J. Sequeda, and M. Wylot (2013) "NoSQL Databases for RDF: An Empirical Evaluation," in Alani et al. [7], pp. 310–325.

[34] "HBase NoSQL Database," http://hbase.apache.org/, [Last accessed July 2014].

[35] (2011), "Couchbase NoSQL Database," http://www.couchbase.com/, [Last accessed July 2014].

[36] "Cassandra NoSQL Database," http://cassandra.apache.org/, [Last accessed July 2014].

[37] (2004), "ARC2," https://github.com/semsol/arc2/wiki, [Last accessed July 2014].

[38] (2006), "BigData," http://www.systap.com/, [Last accessed July 2014].

[39] Gropp, W., E. Lusk, N. Doss, and A. Skjellum (1996) "A High-performance, Portable Implementation of the MPI Message Passing Interface Standard," *Parallel Comput.*, **22**(6), pp. 789–828.
URL http://dx.doi.org/10.1016/0167-8191(96)00024-5

[40] Neumann, T. and G. Weikum (2010) "The RDF-3X engine for scalable management of RDF data," *VLDB J.*, **19**(1), pp. 91–113.

[41] Wilkinson, K. and K. Wilkinson (2006) "Jena property table implementation," in *Scalable Semantic Web Knowledge Base Systems - SSWS*.

[42] Dehaan, D. E. and F. W. Tompa (2007), "Optimal Top-Down Join Enumeration (extended version)," [Last accessed July 2014].

[43] Moerkotte, G. and T. Neumann (2006) "Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees Without Cross Products," in *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB, pp. 930–941.
URL http://dl.acm.org/citation.cfm?id=1182635.1164207

[44] Huang, J., D. J. Abadi, and K. Ren (2011) "Scalable SPARQL Querying of Large RDF Graphs," *PVLDB*, **4**(11), pp. 1123–1134.

[45] Lee, K. and L. Liu (2013) "Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning," *Proc. VLDB Endow.*, **6**(14), pp. 1894–1905. URL `http://dl.acm.org/citation.cfm?id=2556549.2556571`

[46] Lee, K., L. Liu, Y. Tang, Q. Zhang, and Y. Zhou (2013) "Efficient and Customizable Data Partitioning Framework for Distributed Big RDF Data Processing in the Cloud," in *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, pp. 327–334.

[47] "RDF store benchmarking," `http://www.w3.org/wiki/RdfStoreBenchmarking`, [Last accessed July 2014].

[48] "Semantic web challenge 2008. billion triples track," `http://challenge.semanticweb.org/`, [Last accessed July 2014].

[49] Guo, Y., Z. Pan, and J. Heflin (2005) "LUBM: A Benchmark for OWL Knowledge Base Systems," *Semantic Web Journal*, **3**(2-3), pp. 158–182.

[50] Bizer, C. and A. Schultz (2009) "The Berlin SPARQL Benchmark," *Int. J. Semantic Web Inf. Syst.*, **5**(2), pp. 1–24.

[51] Cushing, J. B., J. C. French, and S. Bowers (eds.) (2011) *Scientific and Statistical Database Management - 23rd International Conference, SSDBM 2011, Portland, OR, USA, July 20-22, 2011. Proceedings*, vol. 6809 of *Lecture Notes in Computer Science*, Springer.

Appendix

DBPSB Queries

1	SELECT DISTINCT ?var1 WHERE { <http://dbpedia.org/resource/Akatsi> <http://www.w3.org/2004/02/skos/core#subject> ?var1 . }
2	SELECT DISTINCT ?var WHERE { ?var dbpp:redirect ?var1 . } LIMIT 1000

3	SELECT DISTINCT ?var1 WHERE { { <http://dbpedia.org/resource/Cleaning_Time> dbpp:writer ?var1 . } UNION { <http://dbpedia.org/resource/Cleaning_Time> dbpp:executiveProducer ?var1 . } UNION { <http://dbpedia.org/resource/Cleaning_Time> dbpp:creator ?var1 . } UNION { <http://dbpedia.org/resource/Cleaning_Time> dbpp:starring ?var1 . } UNION { <http://dbpedia.org/resource/Cleaning_Time> dbpp:executiveProducer ?var1 . } UNION { <http://dbpedia.org/resource/Cleaning_Time> dbpp:guest ?var1 . } UNION { <http://dbpedia.org/resource/Cleaning_Time> dbpp:director ?var1 . } UNION { <http://dbpedia.org/resource/Cleaning_Time> dbpp:producer ?var1 . } UNION { <http://dbpedia.org/resource/Cleaning_Time> dbpp:series ?var1 . } }
4	SELECT ?var1 WHERE { <http://dbpedia.org/resource/Bazaar-e-Husn> <http://dbpedia.org/ontology/abstract> ?var1. FILTER langMatches(lang(?var1), 'en')}
5	SELECT ?var6 ?var8 ?var10 ?var4 WHERE {?var4 skos:subject <http://dbpedia.org/resource/Category:McFly> .?var4 foaf:name ?var6 .OPTIONAL {?var4 rdfs:comment ?var8 .FILTER (LANG(?var8) = 'en') .}OPTIONAL {?var4 rdfs:comment ?var10 .FILTER (LANG(?var10) = 'de') .}}

6	SELECT * WHERE {<http://dbpedia.org/resource/Dodgy> ?var0 ?var1.filter(?var0 = dbpedia2:redirect)}
7	SELECT DISTINCT ?var1 WHERE { <http://dbpedia.org/resource/Randy_Brecker> dbpedia2:instrument ?var1 FILTER (langMatches(lang(?var1), 'EN'))}
8	SELECT * WHERE { {<http://dbpedia.org/resource/Cabezamesada> rdfs:comment ?var0. FILTER (lang(?var0) = 'en')} UNION {<http://dbpedia.org/resource/Cabezamesada> foaf:depiction ?var1}UNION { <http://dbpedia.org/resource/Cabezamesada> foaf:homepage ?var2}}
9	SELECT ?var1 WHERE { <http://dbpedia.org/resource/Ryfylke> rdfs:label ?var1 .}