Faculty of Science and Technology

MASTER'S THESIS

Study program/ Specialization: Master in Computer Science	Spring semester, 2011 Open / Restricted access
Writer: Yi Han	…… (Writer's signature)
Faculty supervisor: Professor. PhD. Chunming Rong; PhD Candidate. M.Sc. Tomasz Wiktor Wlodarczyk External supervisor(s):	
Titel of thesis: Parsing and Executing Semantic Queries in a Distributed Environment	
Credits (ECTS): 30	
Key words: Hadoop; Semantic Web; Distributed Computing	Pages: …… + enclosure: …… Stavanger, …….. Date/year

Some sections in this thesis have been published in some publications or are based on sections of previously submitted reports, which authored by the same author of this thesis. Following is the full list of all those sections.

Section 3.3 Converting Queries to Cascalog is based on report of course Project in Computer Science (MID 240) submitted to UiS in 2010 autumn.

Section 4.3.3 Data File Encoding and Section 5.4 Performance Dependent on Data Structure have been submitted as part of the paper “Evaluation of Some Optimization Techniques for Semantic Query Answering on Shared-nothing Architecture [16]” submitted to AINA‐2011 Special Issue of International Journal of Space‐Based and Situated Computing (IJSSC).

Test results in Section 5.3 Performance Dependent on Cluster Configuration have been published in paper “Performance Analysis of Hadoop for Query Processing [13]” at WAINA 2011.

Acknowledgements

I would like to thank Tomasz Wiktor Wlodarczyk, a PhD candidate in Department of Computer Science of UiS, who provided this promising subject to me. Tomasz gave me so much help on the system design and thesis writing. This thesis could not be done without him.

Thank Professor Chunming Rong for his help and supervising this work, and holding a weekly meeting for everyone to exchange ideas.

Contents

List of Figures

List of Tables

Abstract

Semantic data querying has gain more attention recent years. Different technologies have been developed to carry out queries efficiently. However, most of them are focused on improving the data IO performance. This thesis presents our approach of designing an ontology querying system. The system executes query on multiple machines in parallel.

Hadoop is used as the distributed environment framework. Cascalog is used to execute queries on Hadoop. A triple store is implemented in this thesis, which supports access triples from files encoded by UTF-8 or integer encoding. The two encodings and their performance compare are provided and analyzed.

1 Introduction

1.1 Background

Semantic web, being introduced by Tim Berners-Lee, is a set of technologies that make the existing web resources machine understandable. Semantic web resources acts as Meta data of normal web resources (for example, web pages), by describing the information included in a web resource using a machine readable format. It enables applications to access, query and even understand content of resources.

Different languages have been developed to describe information on web resources. For example, the RDF is a widely accepted one based on XML. A file written in language like RDF is called an "ontology" file, containing both concept definition needed to describe the resource and the resource description. By examine ontology files, an applications "understands" information on web resources by mapping it to a supported data model.

1.2 Problem Definition

Technologies have been introduced for ontology querying. However, since the process is very complex, performance could be unacceptable with large data set and complex queries. Different solutions have been proposed on this issue.

One common approach is to save ontology in form of triples in a triple store optimized for data access. The triple store may be distributed or on a server for example a MySQL database. This approach improves performance by having efficient data IO. However, since the query processor still runs on a single machine, it becomes the bottleneck.

1.3 Proposed Solution

Our solution is to distribute the query processor program on multiple machines, i.e. as a distributed system. In this way, we can have multiple query processes running in parallel with part of the work. Hadoop is chosen to be the distributed environment manager. We use Cascalog as middle language in query execution, since it provides a way to query data on Hadoop. That is, when processing a query, it is firstly converted to a Cascalog query and then executed by Cascalog on Hadoop MapReduce service. A triple store is implemented and it can work with Hadoop's native storage HDFS.

1.4 Thesis Organization

The thesis is organized into two parts. In fist part, we introduce the system we developed for the

solution and let the readers have an idea of how it works and the idea behind the system. This part consists of from Chapter 2 to 5. In Chapter 2, we introduce technologies involved in designing and implementing the system. In Chapter 3, the detail of our method is described and analyzed. Chapter 4 presents the design of the system, and at the end we describe how a query is executed by the system step by step with code.

The second part we describe the system performance and some conclusion we get from designing and tuning the system. This part consists of Chapter 5 and 6. In Chapter 5 we describe and analyze the performance tests of the system. And in Chapter 6, we summarize what we have mentioned in this thesis.

2 Background

2.1 Distributed computing

Distributed computing is a system architecture that utilizes multiple machines connected by network working in parallel. It is a way to do complex computing which may need unacceptable long time if ran on a single computer. In some other cases, distributed computing also refers to parallel computing using multiple local processes [6].

Master Slave architecture is commonly used in distributed systems. As shown in following diagram:

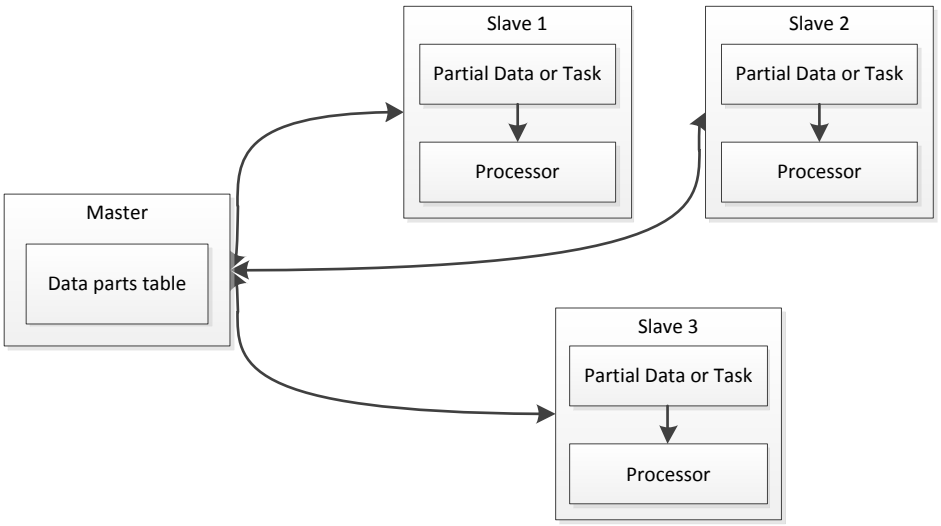

Figure 1 Master Slave Architecture of Distributed Computing Systems

In this architecture, slaves are independent machines that communicate only with the master. On the other hand, the master is responsible for managing slave machines and monitoring job status, and usually there is only one master in such system. Every new job is needed to be submitted to the master before they can be processed by slaves. It then will be divided into several small tasks and assigned to idle slaves. The master monitors the processing status of each task, and collects results.

2.2 Semantic Web

The Semantic Web is a concept introduced by Tim Berners-Lee as a component in Web 3.0. The aim of semantic web is to enable applications to understand information on web resources, in other words, it is “a web of data that can be processed directly and indirectly by machines.” [7]

2.3 Ontology

Ontology is a knowledge representing technique for describing information on web resources. The word “ontology” origins from philosophy, is a study of “the nature of being, existence or reality as such, as well as the basic categories of being and their relations [15]”.

Ontology describe a knowledge domain by define its **terms or objects**, **relationships** and **properties**. Terms are defined as classes. A class can be seen as a predefined data model, it may have inheritance relationship like “sub class of” between other classes. Within a class, it may have properties containing a simple value like string or integer or a complex value, which is an individual (or instance) of a class.

2.4 RDF

RDF (Resource Description Framework) is a ontology describing language. It is one of W3C’s recommendations and has been widely accepted.

RDF is built upon XML, so it is easy to be parsed by machine and read by human. Like XML, data model must be predefined. Data model of an RDF is defined by RDF Schema. RDF Schema is also XML based, it can be used to describe classes, properties and class hierarchy in RDF documents. For example, following XML is a RDF Schema document defining classes and properties describing a vehicle:

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<rdfs:Class rdf:ID="vehicle">
    <rdfs:comment>The class of staff vehicles.</rdfs:comment>
</rdfs:Class>

<rdfs:Class rdf:ID="car">
    <rdfs:comment>The class of cars.</rdfs:comment>
    <rdfs:subClassOf rdf:resource="#vehicle"/>
</rdfs:Class>

<rdfs:Class rdf:ID="bike">
    <rdfs:comment>The class of bikes.</rdfs:comment>
    <rdfs:subClassOf rdf:resource="#vehicle"/>
</rdfs:Class>

<rdfs:Property rdf:ID="wheels">
    <rdfs:comment>Number of wheels.</rdfs:comment>
```

```
    <rdfs:domain rdf:resource="#vehicle "/>
    <rdfs:range rdf:resource=" rdf:resource="http://www.w3.org/2000/01/rdfschema#Literal"/>
</rdfs:Property>

</rdf:RDF>
```

In the RDFS above, a super class “vehicle” is defined to represent a general vehicle; it has property “wheels” indicating the number of wheels on a vehicle. Two classes derive from “vehicle”: “car” and “bike”. A RDF example utilizing this RDFS is shown below:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22rdfsyntaxns#"
        xmlns:sample=”http://www.sampleurl.org/sample”>
    <rdf:Description rdf:about="http://www.sampleurl.org/myVehicles.htm">
    < sample:car rdf:ID=”car1”>
        <sample:wheels >4</sample:wheels >
    </ sample:car >
    < sample:bike rdf:ID=”bike1”>
        <sample:wheels >2</sample:wheels >
     </ sample:bike >
</rdf:Description>
</rdf:RDF>
```

Another way to present RDF data is linked graph. In a linked graph, objects are linked by arcs. An arc represents the relationship between two objects. Above RDF can be represented by graph:

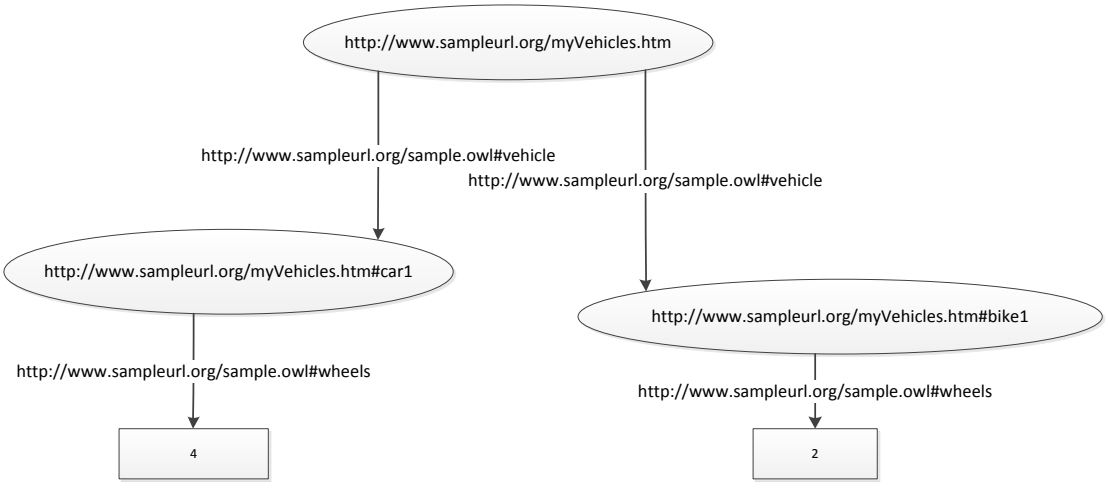

Figure 2 RDF Graph Example

From above graph we can see, RDF model consists of a collection of relationships between two objects. The set of two objects and their relationship is called a triple. The three elements are called subject, object and predicate respectively. To give an example of triples, we can represent Figure 2 with the 4 triples followed:

- {http://www.sampleurl.org/myVehicles.htm, http://www.sampleurl.org/sample.owl#vehicle, http://www.sampleurl.org/myVehicles.htm#car1}
- {http://www.sampleurl.org/myVehicles.htm, http://www.sampleurl.org/sample.owl#vehicle, http://www.sampleurl.org/myVehicles.htm#bike1}
- {http://www.sampleurl.org/myVehicles.htm#car1, http://www.sampleurl.org/sample.owl#wheels, 4}
- {http://www.sampleurl.org/myVehicles.htm#bike1, http://www.sampleurl.org/sample.owl#, 2}

2.5 Triple Store

Triple store is a data store that stores triples. It is often used as storage for ontology data like RDF. Before it can be put in triple store, ontology data firstly have to be converted to triples. A reasoner may be used to ensure all explicit and inexplicit knowledge in ontology has been included in converted triples. By using a triple store, complex ontology can be accessed as triples, making it simple to query and analyze.

Various triple stores have been designed. In "Semantic web marvels in a relational database [1]", Patrick van Bergen provides a step by step tutorial on how to implement a triple store in a relational database. Ching-Long Yeh and Ruei-Feng Lin [2] also demonstrated their approach in designing a triple store using DBMS.

For triple stores that gain publicly acceptance, the triple store included in "Jena2 Database Interface [3]" uses popular database servers - for example MySQL and Oracle – for Jena to store RDF data. Sesame [4] is a RDF processing framework, supports triples persistence on RDMS. 4-store [5] is a RDF database written in C++, supporting remote access and querying RDF data using standard HTTP SPARQL protocol.

2.6 Semantic Web Query Languages

Semantic web query languages are query languages for ontology data. Common languages including SPARQL and SQWRL support ontology defined in RDF Schema and OWL. We will look into the detail of SPARQL and SQWRL in Chapter 3.3.

2.7 Map Reduce

Map reduce is a programming pattern that is popular in data processing systems. It is useful for computing summarized values of grouped data over a large data set. A map reduce process contains two sub routines, map and reduce.

- Map – It takes a key-value pair and produces a collection of key-value pairs. It can be seen as extracting a list of data items from a given data item.

A process does this operation is called a mapper. Since usually a large data set is used as input, it is normal to have multiple mappers doing work in parallel. Parallel mappers may work on different machines and each of them processes part of the input. After all mappers complete, result will be partitioned and input to reduce. The partition is to ensure data items with same key are sent to same reducer (when multiple machines are used in reduce).

- Reduce – It takes result from map, groups data items by keys, generates one data item for each group and adds it to MapReduce result. A process does this operation is called a reducer.

2.8 Hadoop

Hadoop is a distributed computing framework developed by Apache Software Foundation. It provides a number of sub projects to ease the development of distributed applications. Hadoop supports running MapReduce jobs, which provides an efficient way to process big data sets. A distributed file system HDFS (Hadoop Distributed File System) is also included in Hadoop.

3 Executing Semantic Queries on Hadoop

In this chapter, we describe our method for executing semantic queries on Hadoop. The idea is shown in the diagram below:

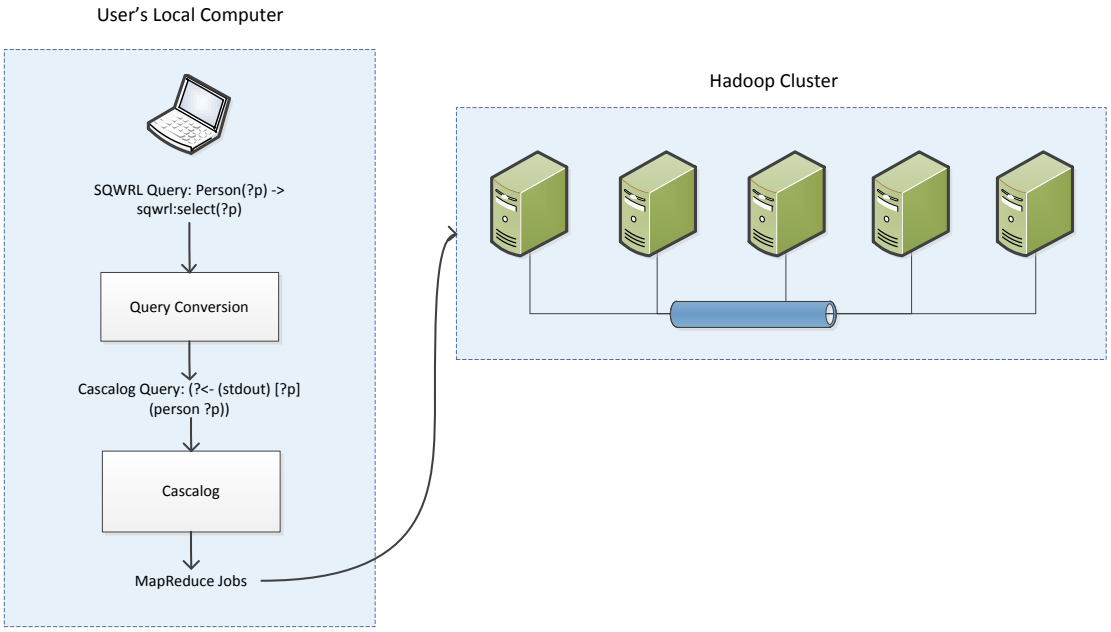

Figure 3 Query Execution Method

1. A supported semantic query, for example a SPARQL and SQWRL query, is converted to a Cascalog query.
2. Executes the query in Cascalog. Internally, Cascalog generates MapReduce jobs from the input query, and execute those jobs on Hadoop.

We describe each step in this chapter. Firstly, we'll introduce the Cascalog.

3.1 Cascalog

Cascalog is a query tool written in Clojure, a functional programming language that built on Java. In turn, Cascalog is running in Java VM.

A Cascalog query to get all adults printing result to console may looks like:

```
(?<- (stdout) [?p] (adult ?p))
```

The query can be divided into three parts,

1. (stdout) – The output sink. An output sink is an object that used to save the query result. The example above uses the “stdout” sink which prints result to console (terminal).
2. [?p] – Output parameter selector. In Cascalog, a data property value (or columns if we regard each data object as a row in a table) is bound to a variable, whose name starts with a question mark “?”. In the example, the adult have only one property (or column), and its value is bound to variable “?p”. When a query gets complex, it may have many variables, and the parameter selector defines which variable should be put in result.
3. (adult ?p) – Predicates. Data source, operations and aggregators are predicates in Cascalog. Predicates define how the query should be carried out and will be executed in the order they are written.

As we can see Cascalog is very expressive and simple. However, we also have other considerations on choosing Cascalog:

1. Easy to convert from SPARQL and other query languages.
   Since Cascalog has similar grammar to SPARQL, the query conversion process could be easy.

2. Strong extendibility.
   Users can write custom predicates to meet their needs, mean while, data source and sink can also be extended to support other data storages for example data tables in HBase. It enables us to support some specific features in semantic query languages, for example user custom functions in SPARQL (it is not supported by system described in this thesis, but Cascalog gives us ability to do so).

3. Run query on multiple machines in parallel
   Cascalog is built on Cascading, which is a data processing framework for Hadoop. This dependency enables Cascalog to execute its query in distributed environment without any modification. Therefore, we can execute semantic queries on multiple machines as long as we can convert them to Cascalog queries.

3.2 Executing a Cascalog Query

Cascalog provides a simple way to write queries but it does not do the execution by itself. A query will then be used to generate a Cascading flow which then will be run by Cascading on Hadoop. Therefore, one can see a Cascalog query as a simple form of a Cascading flow.

“The Cascading processing model is based on a ‘pipes and filters’ metaphor. The developer uses the Cascading API to assemble pipelines that split, merge, group, or join streams of data while applying operations to each data record or groups of records. [10]” In Cascading, a flow is a set of pipe assemblies that connected with multiple sources and a sink. Data comes from sources and goes in to sink. On its way, it goes through the predefined pipe assemblies and gets processed.

3.2.1 From A Cascalog Query to Hadoop MapReduce Jobs

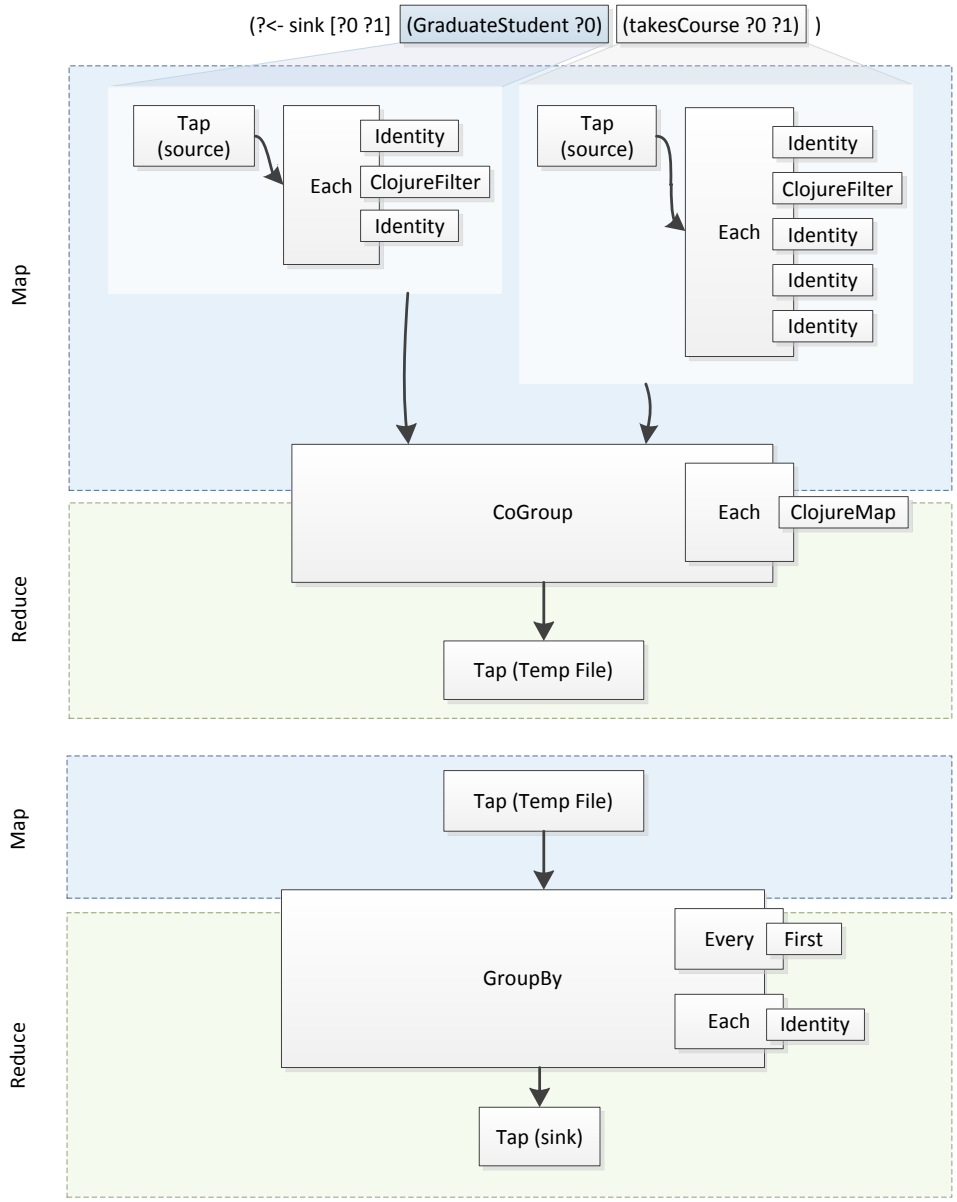

Figure 4 Example of Executing a Cascalog Query Using MapReduce Jobs

Figure 4 shows how Cascalog query "(?<- sink [?0 ?1] (GraduateStudent ?0) (takesCourse ?0 ?1))" is processed using MapReduce. This query finds graduate student names and courses they took.

Firstly, Cascalog parses the query and generates a Cascading Job to run the query. Inside a Cascading Job, there are a collection of linked operations (pipe assemblies and their functions) that processes data. Cascading uses directed graph to represent the linked operations and converts them to MapReduce jobs. In Figure 4, blocks with solid border are Cascading operations. One can see some operations are nested inside another operation. In this case, operations inside will be called according to its parent operation's logic. For example, sub operations in the "Each" operation will be called top down for each input tuple. One or multiple Cascading operations may

be executed in one MapReduce job, and results are saved temporary on HDFS and will be used as input to next MapReduce job.

In the example, two MapReduce jobs are executed to complete the query. In the first MapReduce job, data from two sources are read and filtered to avoid duplicate, respectively. In the reduce phase, a “CoGroup” operation is used to inner join the two data sources together. The result of the first MapReduce job is then saved in a temp HDFS file. The second MapReduce job groups the joined data by all its columns (“?0” and “?1”). The result of the grouping is duplicated entries are removed. The result then is written to the sink Tap as query result.

This example gives an idea of how Cascalog queries are executed in Hadoop. Such process is transparent to end users. However, when it comes to customizing data source and sink Tap, knowing some internal IO procedure of Hadoop is required, and we will introduce it in next section.

3.2.2 Extend Cascalog to Support Custom Triple Store

Tuple is a data structure used by Cascading on data processing. It consists of a set of name-value pairs. A triple can be seen as a tuple with 3 or fewer elements. Therefore, ontology data files like RDF need to be saved as triples before they can be queried. The detail of conversion is described in 3.3, and in this part, we show how Cascalog can be modified to support our triple store.

Cascalog’s custom data source feature is based on Cascading’s Tap. Following diagram shows in Hadoop, how Cascading processes data comes from Tab (blue blocks are components from Hadoop, gray ones are components from Cascading).

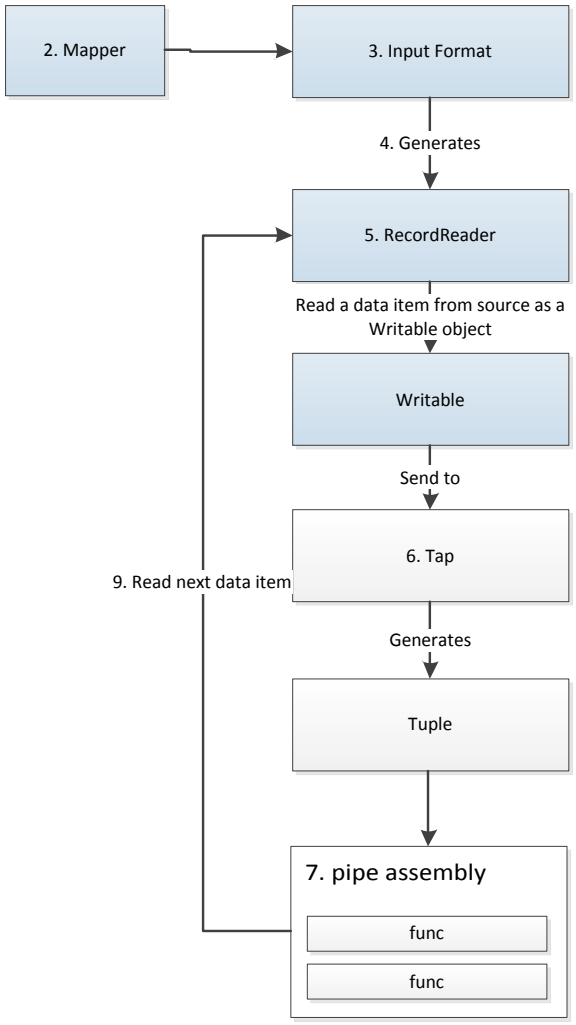

Figure 5 Flow Diagram of Cascalog Data Processing in Hadoop

1. Cascading initializes a Tap for data reading. The Tap configures Hadoop to use its InputFormat implementation as map step data source.
2. Hadoop begins map step.
3. The mapper gets InputFormat from Job Config.
4. The mapper gets a RecordReader from the InputFormat for data reading.
5. The mapper reads one data record, and processes it using the operations configured by Cascading (step 6 to 9).
6. Data record first will be converted to a Tuple object by the custom Tap,
7. Then goes through a collection of Cascading functions to process the tuple.
8. The resulting tuple will be stored in a temp HDFS file waiting to be joined with other job results.
9. If more data record exists, go to step 5.

The Tap object plays a core role in the whole process. It tells the mapper which InputFormat class it should use to read the records from, it also is responsible for converting data record to a Tuple object. In sink process, similar steps are used.

To support a custom data store, following classes should be derived and implemented:

- cascading.tap.Tap: instructs a mapper or a reducer to use the custom InputFormat or OutputFormat, and converts between raw data record and Tuple object.
- org.apache.hadoop.mapred.InputFormat<TKey, TValue>: reads raw data into Hadoop, the data type specified in generic parameter TKey and TValue must be supported by Hadoop's serialization.
- org.apache.hadoop.mapred.OutputFormat<TKey, TValue>: writes raw data to IO stream.

3.3 Converting Queries to Cascalog

Currently the system support converting SQWRL and SPARQL to Cascalog. The conversion process is described in this part. This section is based on report of course *Project in Computer Science (MID 240)* submitted to UiS in 2010 autumn.

3.3.1 Converting SQWRL to Cascalog

3.3.1.1 General Syntax of SQWRL

The syntaxes of Cascalog and SQWRL are similar, both uses function-call-like syntaxes. A SQWRL query has following structure:

```
Rule [^ Rule]* [˚ GroupConstruction [^ GroupConstruction]* [˚ GroupOperation [^ GroupOperation]*]] -> Consequence [^ Consenquence]*
```

A SQWRL statement consists of tail and head. Tail contains rules describing the desired data; head contains "consequences" for data matches the rules. In SQWRL, the head mainly contains output information, such as selecting, sorting and aggregations.

On the other hand, a Cascalog query's structure is:

```
?<- (output sink) [output parameters] (source of data|predicates)
```

Cascalog queries look similar to SQWRL, but differ in some places. The head part in Cascalog comes first, with all data columns or parameters needed to be included in the output. Then follow the "rules", which in this case, the predicates. Each predicate is a Clojure function or macro, thus makes it easy to convert SQWRL to Cascalog. Data sorting and aggregation is regarded as predications too, it means when converting, tricks are needed on the Consequences.

3.3.1.2 Basic Conversion Idea

Since both query syntax has monotony attribute and uses functions to refine/filter data, it is possible to map each SQWRL rule to a Cascalog predicate. For basic rules without collections, converting procedure is listed below:

a) Maps each rule in tail to Cascalog predicates.
b) Maps consequences except *select* and selectDistinct to Cascalog predicates (generates a (:distinct false) predicate if no selectDistinct is found), assigns each predicate a new parameter and append these predicates to the previous ones.
c) Finds all select and selectDistinct consequences and extract the parameters, appends the new parameters in step b.
d) Puts the predicates and the output parameters together.

3.3.1.3 Basic built-in transforms

Generally, following operations (actions) are used to transform SQWRL rules and consequences. To make them shorter, we also gave each action a name:

a) Map -> *Map*
b) Assign parameter -> *AssignPara*
c) Append parameter in output parameters list -> *AppendParaToOutput*
d) Append rule or consequense in predicate list -> *AppendToPredicates*
e) Ignore -> *Ignore*
f) Not supported/Error -> *Error*

Following table lists operations of transforming basic SQWRL built-ins to Cascalog predicates, excluding collection built-ins.

Built-In name	Ex	Cascalog ex	Operations	Comments
swrlb:selectDistinct		N/A	AppendParaToOutput	
swrlb:select		N/A	AppendParaToOutput	
swrlb:count	sqwrl:count(?p)	(c/count ?pOut)	Map, AssignPara, AppendParaToOutput, AppendToPredicates	The group is formed automatically by specifying output parameters, and the count function in Cascalog just count the number of items in each group, thus no input parameter is needed in count.
swrlb:columnNames		N/A?		
swrlb:orderBy	sqwrl:orderBy(?p)	(:sort ?p)	Map, AppendToPredicates	
swrlb:orderByDescer	sqwrl:orderByDesc	(:sort ?p) (:reverse t	Map, AppendToPredicates	
swrlb:min	sqwrl:min(?p)	(c/min ?n :> ?pOut)	Map, AssignPara, AppendParaToOutput, AppendToPredicates	
swrlb:max	sqwrl:max(?p)	(c/max ?n :> ?pOut)	Map, AssignPara, AppendParaToOutput, AppendToPredicates	
swrlb:avg	sqwrl:avg(?p)	(c/avg ?n :> ?pOut)	Map, AssignPara, AppendParaToOutput, AppendToPredicates	
swrlb:sum	sqwrl:sum(?p)	(c/sum ?n :> ?pOut)	Map, AssignPara, AppendParaToOutput, AppendToPredicates	
swrlb:median		N/A?		

Table 1 Operations of Transforming Basic SQWRL Built-Ins

3.3.1.4 Example

1. Following query selects names and ages of persons older than 17:

Person(?p) ∧ hasAge(?p, ?age) ∧ swrlb:greaterThan(?age, 17) → swrlb:select(?p)

Firstly all rules are mapped to Cascalog predicates according to **Error! Reference source not ound.**:

Person(?p) => (person ?p)
hasAge(?p, ?age) => (hasAge ?p ?age)
swrlb:greaterThan(?age, 17) => (> ?age 17)
swrlb:select(?p) => [?p]

Combing them all together with selector at the front, we get the Cascalog query:
(?<- (stdout) [?p] (person ?p) (hasAge ?p ?age) (> ?age 17))

2. Following query queries first two names in the result:

Person(?p) ∧ hasName(?p, ?name) → sqwrl:select(?p, ?name) ∧ sqwrl:columnNames("Person Names") ∧ sqwrl:limit(2)

Converted predicates are:

Person(?p) => (person ?p)
hasName(?p, ?name) => (hasName ?p ?name)
sqwrl:columnNames("Person Names") => *Not Supported by Cascalog*
sqwrl:limit(2) => (c/limit [2] ?p ?name :> ?p-out ?name-out)

The last predicate creates two new variables “?p-out” and “?name-out”, they should be the variables in the output list rather “?p” and “?name”. Such change is needed to be tracked to reflect in the output list. The resulting Cascalog query is:

(?<- (stdout) [?p-out ?name-out] (person ?p) (hasName ?p ?name) (c/limit [2] ?p ?name :> ?p-out ?name-out)

3.3.2 Converting SPARQL to Cascalog

3.3.2.1 General Syntax of SPARQL

A SPARQL query consists of four parts: prefix declaration, query form, where clause and modifiers.

Generally, a SPARQL query has following structure:

```
SELECT <argument list>
WHERE
{
    [[OPTIONAL] <subject> <predicate> <object>[, <object>]*
    [; <predicate> <object> [, <object>]* ]*.]+
}
ORDER BY <argument>
```

A formal grammar has been defined as:

```
'SELECT' ( 'DISTINCT' | 'REDUCED' )? ( Var+ | '*' ) DatasetClause* WhereClause SolutionModifier
```

Full grammar description can be found at http://www.w3.org/TR/rdf-sparql-query/#grammar [2].

3.3.2.2 Basic idea

Apart from SELECT which returns the full result set matches the rules, three other query forms are also provided: CONSTRUCT, ASK and DESCRIBE, which provide additional formatting based on the result set. Therefore, CONSTRUCT, ASK and DESCRIBE can be implemented by using a custom Java method transforming query result from a corresponding SELECT query. We now describe

how to transform SELECT sub clauses to Cascalog.

WHERE clause

A WHERE clause may have following elements:

- Triple or triples list
- Optional keyword: inner clause can have the same structure as WHERE clause.
- Union keyword: inner clause can have the same structure as WHERE clause.
- One or multiple filters from following
  - BracketedExpression
  - BuiltInCall
  - FunctionCall

For triple or triple lists, it can be transformed to Cascalog data source generators, except in the case where the "subject" element is also a variable, for example:

```
PREFIX example: http://swrl.stanford.edu/3.4.4/SQWRLExamples.owl# [3]

SELECT ?type ?relation ?subType
WHERE { ?type ?relation ?subType}
```

The query above lists all triples, it cannot be done with Cascalog since the query does not specify a target object type as data source generator. A reasoner is needed to include all possible object types in the query.

"Optional" indicates an optional query criterion. In different cases, it needs to be converted to different Cascalog predicates:

- Only tuples matching included, for example

  ```
  OPTIONAL { ?p a example:Person. ?p example:hasAge ?a. }
  ```

  It can be converted using outer joins.
- In other cases, use sub-query to represent the optional part, and outer join it in the main query, previous query items whose variables be referenced in this optional clause should also be included in the sub-query.
- For better performance, complex optional clause with only one level (no {} included) can be optimized by removing all FILTER parts since it is nonsense in a OPTIONAL context, for example:

```
PREFIX example: <http://swrl.stanford.edu/3.4.4/SQWRLExamples.owl#>

select ?p    ?d ?c
where { ?p a example:Male.
```

```
?p example:hasDOB ?d.
?p example:hasName ?c.
OPTIONAL{FILTER (?c = 'Alice')}
}
```

And if left parts are all triple matches, approach one can be used, or use the second approach.

UNION clauses can be converted into Cascalog sub-queries and use the union function to union all the results together.

Filters can be treated as functions in Cascalog, result of a filter should be a Boolean value. Filter clause can have one or more conditional expressions, connected with logical operators for example && and ||.

Filter conditions can be surrounded with or without brackets, in cases where only one built-in functions are used, brackets can be omitted:

```
FILTER Bound(?d)
```

In other cases, brackets must be written:

```
FILTER (?a > 10)
FILTER (Bound(?d) && Bound(?d))
```

Several built-in filters provided:

Filter	Description	SPARQL Example	Cascalog Example
STR	“Returns the lexical form of ltrl (a literal); returns the codepoint representation of rsrc (an IRI). This is useful for examining parts of an IRI, for instance, the host-name. (W3C)”	FILTER regex(str(?mbox), "@work.example")	Custom function.
LANG	Returns the language tag.	FILTER (lang(?name) = "ES")	Not supported.
LANGMATCHES	“Returns true if language-tag	FILTER langMatches(lang(?title), "FR")	Not supported.

	(first argument) matches language-range (second argument)" (W3C).		
DATATYPE	Returns the datatype IRI.	FILTER (datatype(?shoeSize) = xsd:integer)	
BOUND	Tests if a variable is bound to a value	FILTER (bound(?date)) }	
sameTerm	Returns TRUE if term1 and term2 are the same RDF term.	FILTER (sameTerm(?mbox1, ?mbox2) && !sameTerm(?name1, ?name2))	
isIRI	Returns true is term is an IRI.	FILTER isIRI(?mbox)	Custom function.
isBLANK	Tests if a variable is a blank node.	FILTER isBlank(?c)	
isLITERAL	Tests if a variable is a literal.	FILTER isLiteral(?mbox)	
REGEX			

Table 2 Basic SPARQL Built-ins

Besides, logical and math operators also supported, including ||, &&, =, !=, <, >, <=, >=, +, -, *, /.

Modifiers

Modifiers in SPARQL are very straight forward, and can be mapped to Cascalog predicates directly.

Modifier	Description	SPARQL Example	Cascalog Example
Order By	Appear next to the Where clause.	ORDER BY ?p	Cascalog does not support final result row order, it only support ordering rows before sending to the aggregators.
Projection	Projection part is the variables list in SELECT clause. It can have variable	SELECT * SELECT ?p	[?p] Cascalog does not support "*" symbol,

	name or “*” representing all variables.		so a full variable list should be used.
Distinct		SELECT DISTINCT ?name	(:distinct true)
Reduced			
Offset	“OFFSET causes the solutions generated to start after the specified number of solutions. An OFFSET of zero has no effect. (W3C) ”	PREFIX foaf: <http://xmlns.com/foaf/0.1/> SELECT ?name WHERE { ?x foaf:name ?name } ORDER BY ?name LIMIT 5 OFFSET 10	Offset and limit can be implemented in Cascalog by introducing a user defined aggregator, which takes all items in a solution and returns the items required. Such aggregator must be put after all other predicates.
Limit		PREFIX foaf: <http://xmlns.com/foaf/0.1/> SELECT ?name WHERE { ?x foaf:name ?name } LIMIT 20	

Table 3 Some SPARQL Filters and Their Conversion Method

Conversion steps

Since SPARQL has good organization on query elements, and grouping is not supported, it is easy to express queries with the exactly the same meaning in Cascalog. Steps of conversion is as followed:

1. Maps all elements in Where clause
   a) Patterns: maps to data source predicate in Cascalog (e.g. (person ?p)).
   b) Patterns with subject as an argument (e.g. ?a ?b “Person”), an error should be thrown.
   c) Simple optional patterns: maps to outer join data source predicate (e.g. (person !!p)).
   d) Complex optional clauses: do as described in 2.1
   e) Filters: mapping filter functions as Cascalog or Clojure Boolean predicates.
2. Maps the DISTINCT or REDUCE modifier.
3. Maps OFFSET and LIMIT clauses into Cascalog aggregators and append the invocations in the predicates list.
4. Scans and validates all arguments used in the predicates.
5. Renders the output arguments list according to the SELECT clause.

3.3.2.3 Example

Following query selects title of products with price lower than 30.5:

```
PREFIX    dc:    <http://purl.org/dc/elements/1.1/>
PREFIX    ns:    <http://example.org/ns#>
SELECT    ?title ?price
WHERE      { ?x ns:price ?price .
             FILTER (?price < 30.5)
             ?x dc:title ?title . }
```

We follow these steps to convert it to Cascalog query. Firstly, statements in WHERE clause are converted to Cascalog predicates:

```
?x ns:price ?price => (price ?x ?price)
FILTER (?price < 30.5) => (< ?price 30.5)
?x dc:title ?title => (title ?x ?title)
```

Since we do not introduce new variable, ones in the SELECT clause can be used in the output list directly:

```
SELECT    ?title ?price => ?title, ?price
```

The resulting query is:

```
(?<- (stdout) [?title, ?price] (price ?x ?price) (< ?price 30.5) (title ?x ?title))
```

4 System Design and Implementation

The system is a distributed ontology querying system with a simple console user interface. It is written in Java (SE 6) and built upon Hadoop and Cascalog. In this chapter, we first describe the design of the system and then present components structure and their interactions. At last, we present important codes by going through a simple system routine.

4.1 System Features

The system provides two major features -- triple storage and query.

- **Triple Storage.** Ontology needs to be converted to triples and imported into system before it can be queried. The conversion is manually done by users or a third party tool. During importing, triples optionally go through a reasoner for materialization[1]. Imported triples will be stored in plain text or integer encoded files on specified distributed storage, for example Amazon S3 or HDFS (Hadoop File System).
- **Query.** Users can use one of the supported query languages – SPARQL or SQAWRL to query imported triples. The system monitors the querying process and stores both query result and statics on user's local machine.

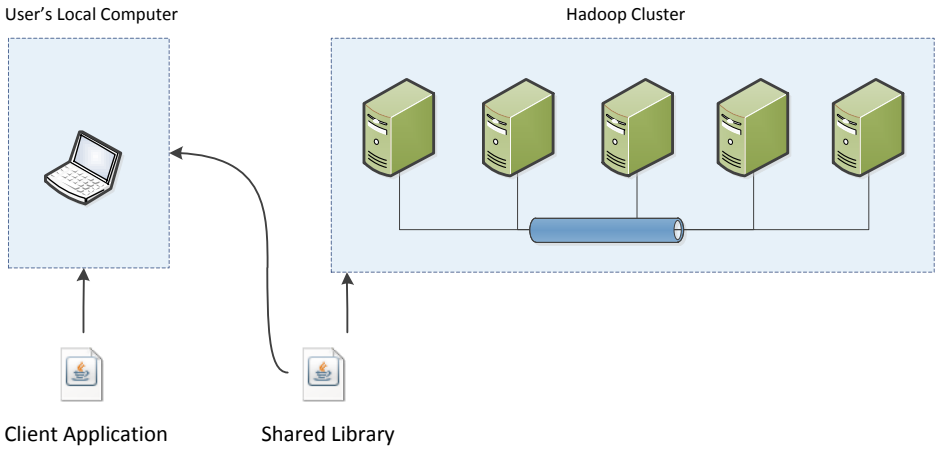

Figure 6 System Deployment

Figure 6 shows the deployment of the system. The system will be deployed on two set of machines, users' local computer and Hadoop cluster.

- Users' Local Computer
  Users use their local computers to interact with the system, for example submit queries. Both user client application and shared library are needed to be deployed on user side. The

[1] A way to extract hidden information (triples) from a given triple. For example, class "Vehicle" has sub-class "Car" and "Bike", materialization on a car individual can produce a vehicle individual since Car is sub-class of Vehicle. However, materialization is not in scope of this thesis

user client is a Java console UI application that takes a user's request and calls the Hadoop cluster to fulfill it. It is also responsible for retrieving process result and display the result to user.

- Hadoop Cluster
  Our system executes queries on Hadoop Cluster. On the cluster's master machine, system's shared library is needed to be deployed, which is used to execute Cascalog sources in Hadoop and access the triple store.

  There is no server-side for Triple Store. To access triples, HDFS API is used to read or write files on Hadoop Cluster. Triple Store can be accessed on both client side and Hadoop Cluster.

4.2 System Architecture

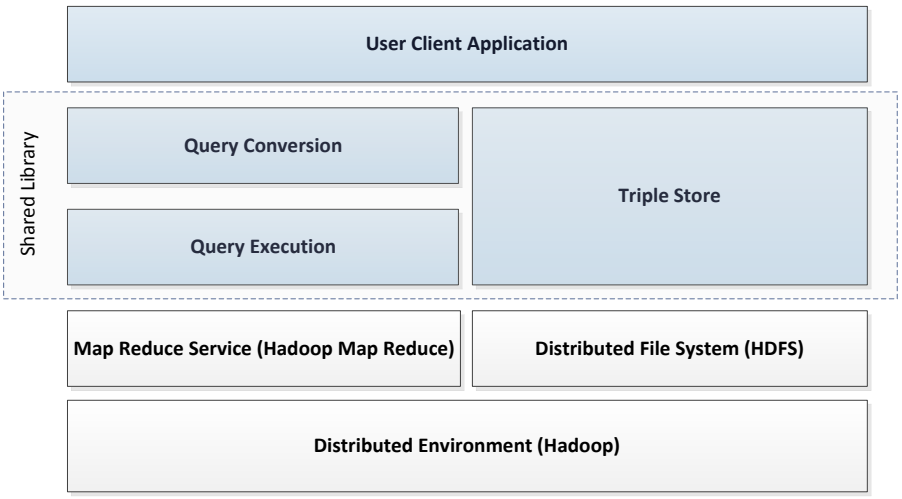

Figure 7 System Architecture

Figure 7 shows the high level system architecture. To make the design simple and extendable, the use of Hadoop is hidden inside lower level components (Query Execution and Triple Store), which also makes it possible to be installed and run locally. Additionally, there is no clear boundary of client side and server side components, both sides have a full package deployed, but only a few classes is used.

The system consists of four major components (blocks with blue background) which are listed below. Beside the User Client Application, the other three ones are members of the Shared Library:

- Query Conversion – The component is responsible for converting semantic queries (SPARQL or SWRL) to Cascalog source files. The component can be easily extended to support more query languages, and it is independent from other components.
- Triple Store – A triple store implementation that can be used on any file systems, locally or distributed. It acts as a backend data store for system to query. Currently the triple store supports following file systems: HDFS, Amazon S3 and Java local file system.

- Query Execution – The component that executes Cascalog queries on Hadoop, monitors the query status and collects the result. The query execution component relays on a map reduce service, currently we use Hadoop MapReduce. To carry out the query, it also needs to access the triple store or other data source.
- User Client Application – UCA is responsible for interact with the user and provides him information on what is happing. It is the entrance of the client side application, it then calls other components fulfill the user's request.

We will explain the four components bottom up. The User Client Application will not be described since it is very simple and no special design has been made on it.

4.3 Triple Store

The Triple Store component is responsible for storing triples on persistent storage, in the system, it acts like a data source for the Query Execution component to complete query requests. The Triple Store benefits the system by providing Triple Store abstraction hiding the internal encoding algorithm and file management. The main goal for the triple store is to provide fast data appending and reading, which is the two main functions used by the Query Execution component. Data files managed by Triple Store are also made portable and independent from other components for easier data backup and restore.

4.3.1 Triple Table

To provide a simple data access model, we use tables to organize triples by grouping them by subject, predicate or object. In this way, a triple table stores triples with at most two columns since the other one can be represented by the table name.

For example, if following triples needs to be uploaded:

{http://www.someuri.org/objects/mycar1, classOf, http://www.someuri.org/Car}
{http://www.someuri.org/objects/mycar2, classOf, http://www.someuri.org/Car}
{http://www.someuri.org/objects/mycar2, http://www.someuri.org/hasInsurance, True}
{http://www.someuri.org/objects/mycar3, classOf, http://www.someuri.org/Car}
{http://www.someuri.org/objects/mycar3, http://www.someuri.org/hasInsurance, False}

The five triples contains information of 3 cars, mycar1, mycar2 and mycar3, the last two cars also has values in their "hasInsurance" property, indicates mycar2 has insurance and the other one does not. To add these triples to the system, following steps should be made:

1. Groups triples by triple elements
2. Initializes new Triple Tables for every group if not exist
3. Append triples to the new table

Currently, the first step is done manually by users. Selecting the triple element to group by is like choosing index key column for a database table, which will directly affect the query performance, for data in that table can be read directly without filtering. It is recommended to group by each triple element respectively, although some groups may have only one triple and resulting tables like “classOf” may never be queried. However, it won’t affect the performance. In above example, we created two tables, “Car” and “hasInsurance”. The resulting tables are:

Subject
<http://www.someuri.org/objects/mycar1>
<http://www.someuri.org/objects/mycar2>
<http://www.someuri.org/objects/mycar3>

Table 4 Table “http://www.someuri.org/Car” (Object)

Subject	Object
<http://www.someuri.org/objects/mycar2>	True
<http://www.someuri.org/objects/mycar3>	False

Table 5 Table “http://www.someuri.org/hasInsurance” (Predicate)

One may see that in Table 4 we also ignored the “classOf” element. The reason is when querying individuals of a data type in semantic queries, the predicate element is always ignored. For example if we want to query all persons in ontology, it can be written in SQWRL as “person(?p)” instead of “classOf(person, ?p)”.

Triple store tracks the original triple position of cell values and the grouped value, so the original three-element-triples can be restored. The grouped value is saved as a part of table name.

4.3.2 Data Files

Triple tables are saved as files. Each table may have multiple files containing partial table data. In other words, data is stored in file splits. This is due to the requirement of the general file system support.

Since we do not limit the system to Hadoop, it must support different file systems. For a Triple Store used in data query, it relays heavily on the following file system features:

- File appending. Adding a triple to system may end up appending data to a file. File appending is not supported for some reason by file systems like S3N. Lack of appending support could bring serious performance issue. Imagine if a table already has 10 GB data, and we need to append 1 KB data, then we need to download a 10 GB file and append 1 KB to it and then upload it back to S3N. Its performance may totally unacceptable and on the other hand, it may cost much money on Amazon S3N data transfer.
- Serial file reading. Processing a query needs to scan the whole table from beginning to the end, jumping back in a data stream in rare in our system.

Storing files in splits can solve the problem of some file systems do not allow file appending. It also enables bulk deleting by uploads if split data is well partitioned. It may add some overhead on file reading because of the additional file IO requests, but the impact may not be significant unless we have too many splits. Plus, when executing a query using MapReduce in distributed environment, having multiple splits make it easy to partition data for mappers.

Two rules are used to decide when to make a new split:

- **New uploads.** Every new upload starts with a new split. This ensures each data file can have a unique upload ID and data in one upload will not affect the other.
- **When file size reaches a fixed length**. Sometimes due to poor network condition or bad processing performance, data reading may failed by timeout. The system will have to recreate the connection and skip data to previous read location. Having limited data file size may reduce the possibility of IO interrupt caused by timeout or other network error.

Currently the system supports local file system, Amazon S3N and Hadoop File System (HDFS).

4.3.3 Data File Encoding

Triples in a table will have to be written to files with encoding. A good encoding algorithm can result smaller data files which in turn, makes query more efficiency. However, the smaller the data is, the more complex the algorithm will be which in turn, more CPU time will be cost. A tradeoff is needed here to decide an appropriate algorithm. We implemented two encodings in the system to study the aspects that affect query performance. Their performance is presented in Chapter 5.4.

This section has been submitted in paper “Evaluation of Some Optimization Techniques for Semantic Query Answering on Shared-nothing Architecture [16]” submitted to AINA‐2011 Special Issue of International Journal of Space‐Based and Situated Computing (IJSSC).

4.3.3.1 Plain Text Encoding

Plain text encoding saves triples as plain text with UTF-8 encoding. It is the fastest way to read and save triples. In both reading and writing process, the only operation is to decode bytes to chars, and put them into character arrays to form a string. Since we use URI to represent triple elements, which uses symbols in ASCII, decoding from bytes to chars requires just a type cast. However, since an ASCII char in UTF-8 takes 1 byte, this encoding may produce large data files, which in turn will affect query performance.

The plain text encoding is not integrated into the Triple Store component. Currently, we use Cascalog’s built-in taps for data input and output, which directly reads and writes manually managed data files. The integration may be done in future versions.

Performance tests for queries on plain text data files can be found in Chapter 5.3

4.3.3.2 Integer Encoding

Integer Encoding replaces triple element strings with integers and stores integers, instead of strings, to data files. Since we only use positive numbers, unsigned bits are used to store a binary integer.

Integer encoding benefits the system in following aspects:

- Efficient Memory Usage. In Java, each character takes 2 bytes in memory, for a 40-character-string, it takes 80 bytes. When using integer encoding, normally 2 integers are used to represent a 32 – 64 bit unsigned integer. In this case, only 8 bytes are used -- 10 times smaller than strings.
- Smaller Files and Shorter Loading Time. To handle bigger data, we use at least 8 bytes for each integer. Compared with original strings, which usually takes 40 bytes (ASCII encoded, 1 byte per character), is about 5 times smaller. Smaller data takes shorter time to read, thus integer encoded files can be loaded much faster by Hadoop.

To make it more clear, the integer encoding is integrated into the Triple Store component, and the system accesses integer encoded data files through Triple Store. Performance compare between plain text encoding and integer encoding are presented in Chapter 5.4.

In Memory Data Structure

After data being read to memory, it needs to be held in some data structure. Such data structure should be easy to compare and use small amount of memory. Two data structures are used for integer encoding and their performances are presented and analyzed in Chapter 5.4. In this section, we describe them respectively.

Integer Array

With integer encoding, bits read from data source are unsigned binary integers. When stored in an integer array, they are grouped by every 32 bits and each group is then converted to a java.lang.Integer object. For example, the integer 70370891661318 is stored as two integers [16384, -2147483642]:

Number		70370891661318								
Bits	Index	63	62-47	46	45 - 32	31 - 4	3	2	1	0
	Value			1	0…	1000…	0	1	1	0
Elements of Integer Array		16384				-2147483642				

Table 6 Representation of 70370891661318 using Unsigned Bits and Integer Array

Since the bits are unsigned, a two-element java.lang.Integer array can represent integers from 0 to $2^{64} - 1$. This is sufficient for most cases. However, the array length can be increased for bigger values.

Bytes Array

For test purpose, we introduce bytes array structure. It is similar to the previous one, but reads data to memory as bytes. In other words, it reads every 8 bits at a time. For example, if loading a 64 bits integer from a file, an 8-byte-array will be generated instead of a 2-integer-array.

File Format

Each encoded triple will be stored in files with following format:

3 bytes header:

- 2 bytes: record state (active or inactive). 2 bytes are used since initially a short is used to represent a record state. It is too large for a state value, but since tests have already been carried out with this configuration, we keep it as 2 bytes in this version.
- 1 byte: record cells count. Since we use each record to store one triple which has at most 3 elements, the value will be between 0 and 4.

2 + N bytes for each cell:

- 2 bytes: cell length (in bytes)
- N bytes: cell value.

Therefore, if all integer values can be represented by 64 unsigned bits,

- For a triple having one element (usually it is object), 3 + 2 + 2 * 4 = 15 bytes are used.
- For a triple having two elements (object and subject), 3 + 2 * (2 + 2 * 4) = 23 bytes are used.

String-Integer Mappings Dictionary

To make the encoding reversible, we must store the original triple element strings. The system maintains a dictionary containing all string - integer pairs and can be loaded on demand. However, since data could be very large, the dictionary may be too big to be kept in Java's heap. To solve this problem, we divided the dictionary into parts and did some optimization to make it work for big data sets.

Grouping

Instead of uses continuous integers as encoding result, the encoder generates a 64-bit unsigned integer from a string's hash code. Following steps are used in generation. Firstly, strings are grouped by the highest 11 bits of their hash codes. Those 11 bits are used as the group index,

which then becomes the first 32 bits of the generated integer (the other 22 bits are 0-value-paddings). The number of strings in the group when a string is added becomes the second 32 bits of the generated integer.

For example, when encoding string “http://test.uri”, we first get its hash code: -735391198, whose first 11 bits are: 11010100001.

Assume there are already 2 strings with the same first 11 bits in their hash codes in the dictionary, then the new one’s index in group is 2, which is 10 in binary. Combining the two indexes’ bits, we get the final encoded integer for the string:

00000000000000000000011010100001 00000000000000000000000000000010

In integer array, it is:

{ 1697, 2 }

Items in the same group are stored in a linked list to provide fast appending and removing. After applying this to the mappings dictionary, its structure looks like a traditional hash table:

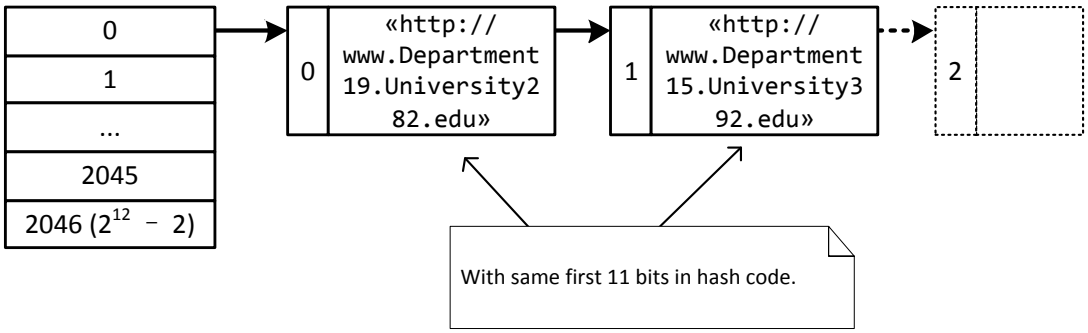

Figure 8 Structure of Mappings Dictionary

Expected Performance

Using this structure, time cost by appending a new string depends on the number of strings in the same group (it needs to compared with all exists strings in group to avoid duplicate). While using a list to store all string mappings, the appending time depends on the number of strings in the list. If we assume the hash function of “java.lang.String” generates uniformly distributed hash codes, for a mappings dictionary already has N strings added, appending a new string cost time:

- Using structure described above, $T = N / (2^{12} – 1)$
- Using list, $T = N$

Therefore, this structure provides faster searching and appending than using a list.

On-Demand Loading

To avoid memory leak, mappings dictionary groups are loaded on-demand. Thanks to the hash table structure, group index can be easily computed from a given string.

We limited the number of mappings in memory to 1040000. This number is suggested by tests, it is close to the value that can cause java heap overflow (in cases when too many long strings are loaded, it may still cause overflow). When this number exceed, the system will have to remove some groups from the memory.

4.3.4 Component Design

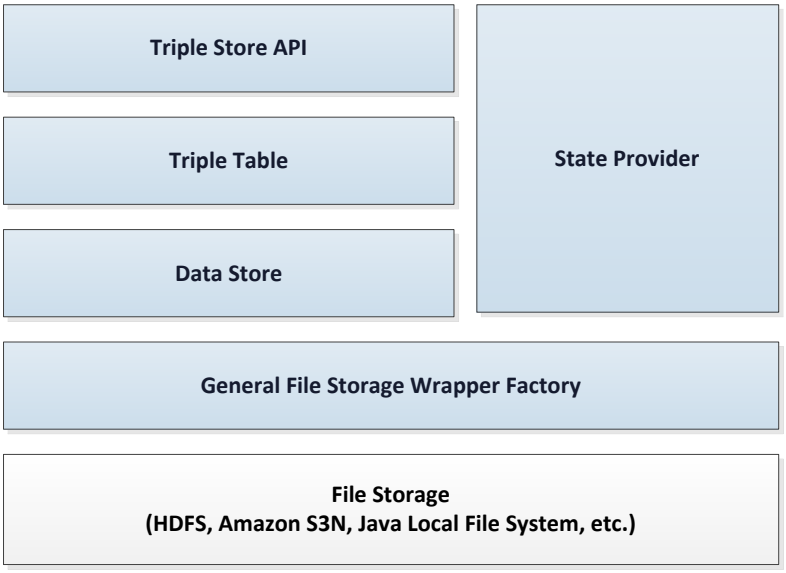

Figure 9 Triple Store Components

Figure 9 shows high level components of the triple store. Same as previous block figures, blue ones are components from the system.

- **Triple Store API**
  Provides interfaces for other components to access triples.

- **Triple Table**
  The triple table provides table structure for data access. As described previously, a triple table itself represents a triple element, so data saved in it is not a complete triple. In this case, we call each data item a “triple row”. Elements (or cells) in a triple row are stored in their original form – strings.

  Triple table uses a data store to save triple rows. An encoder will be used to encode cells with string values to bytes, or convert bytes back to strings. Therefore, elements in a triple row will have to be converted to bytes when interact with the data store. To distinguish the two access levels, the byte version of a triple row is called a “data row”.

- **Data Store**
  A data store is a byte-oriented data table manager, which saves bytes in table structure. For every triple table, a unique data store is used to access table with data in bytes. However, a

data store is not only for storing triples, it is general enough to store any data that can be converted to a collection of bytes array.

Data Store may employ a file split manager to manage data splits. The split manager is responsible for locating a file split with a given row index, and it needs to keep track of row index range in each splits.

- **General File Storage Wrapper Factory**
  The general file storage wrapper factory is for providing file system API wrappers which implements a well-known file system access interface. The file storage wrapper provides methods that enable access files in a general manner. All file systems wrapped is expected to have the same behavior as described in 4.3.2.

- **State Provider**
  Some components may need to resume from their previous state when being initialized. For example, a DataStore needs to know previously created data files to provide data access function. Normally, states will be saved to files and loaded back to memory by the component itself. However, this may result hard and incomplete management of system files and component life-cycles. Therefore, states provider is used to persist component internal states.

  A state is defined as a string-object pair, with the string as the name, and the object as the state value. Every object may have infinite number of states as long as it won't cause memory leak (we currently do not have limit on memory usage). State provider sees states as private properties to its owners. Therefore, an object can only access states created by itself. State provider groups states belong to the same object into a State Bag. To achieve this, each state owner needs to be uniquely identified. The approach we use to identify an object is to give every object a string based name, objects with same name, class and package name is regarded as same objects.

  The actual storage being used to save states is configurable and transparent to the state owners. It could be file system, memory, or database. If states are stored in files, the *General File Storage Wrapper Factory* is used to provide file access.

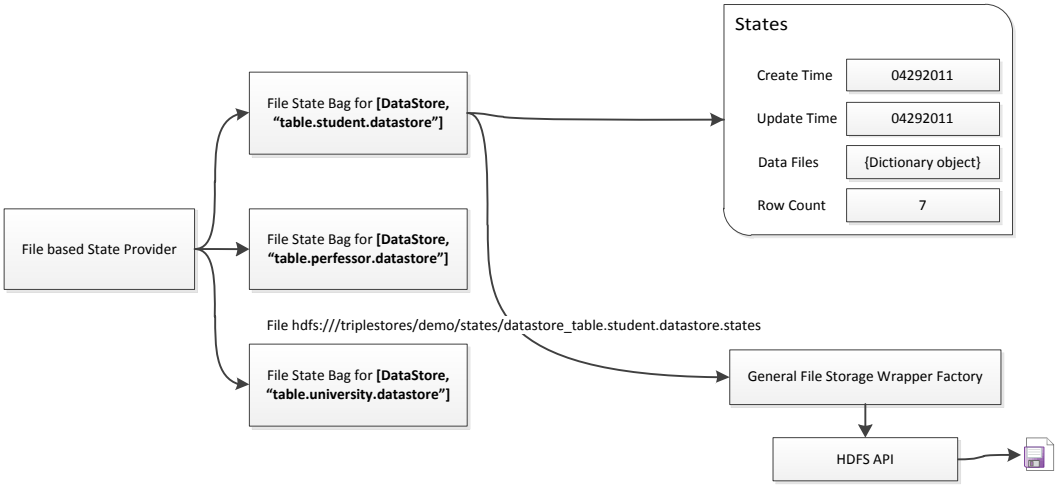

Figure 10 States Provider Demo

Figure 10 presents an example of states management. The states provider used in demo saves states to files. It now has states from three objects, and all of them are of class "DataStore". Object "table.student.datastore" is the data store for triple table "student". It has four states in its states bag. Every time the states bag is saved or refreshed, the state provider will use the file URI to write or read state data from the file. In this process, the HDFS API is used to access file on HDFS.

The state provider benefits the system in following ways:

- Faster development. Using states provider, objects does not need to care about states persistence anymore, which reduces the development work.
- Portable system data. States provider provides a way to extract objects' internal states and manage those states in one place. It is possible to configure the states provider to save all states in files under one directory. In this way, we can backup system data by copying all files in that directory, and copy them back if we need to restore system back to a previous state.
- Transaction ready. Although transaction support is not supported by now, it is useful when multiple processes are accessing a same triple store. To have transaction support in a distributed system, it is important to have distributed locks on object states. States provider is a good place to provide this support since 1) its responsibility -- manages objects' states – is a super set of the state lock feature and 2) it is used by most system components, so no more dependency introduced.

4.4 Query Conversion

The Query Conversion component provides a general query parsing and converting model. By now, it supports converting SPARQL and SQWRL to Cascalog.

A query will be parsed to a general query model, which represents a query by elements. The model is only used for query representation, it does not contain methods related to real execution. Following class diagram shows the model definition:

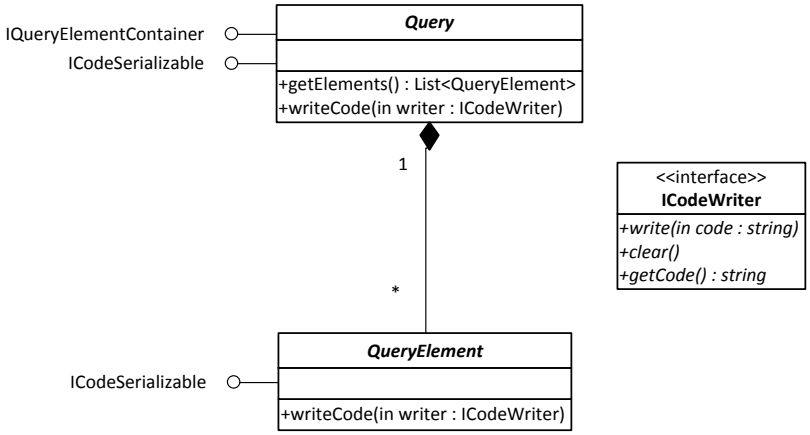

Figure 11 UML Diagram of Query Model

- **Query** represents a query. Each query language should have its own implementation derived from this class to represent queries written in that language.
- **QueryElement** represents an item from the query. QueryElement can be nested to represent a layered relationship. Each query language should create its own set of query element types by deriving from the QueryElement class.
- **ICodeWriter**. An interface whose implantation can be used to write query strings to an output.

For example, SQWRL query "Person(?p) ^ Gender(?p, <male>) -> sqwrl:select(?p)" will be parsed to following structure:

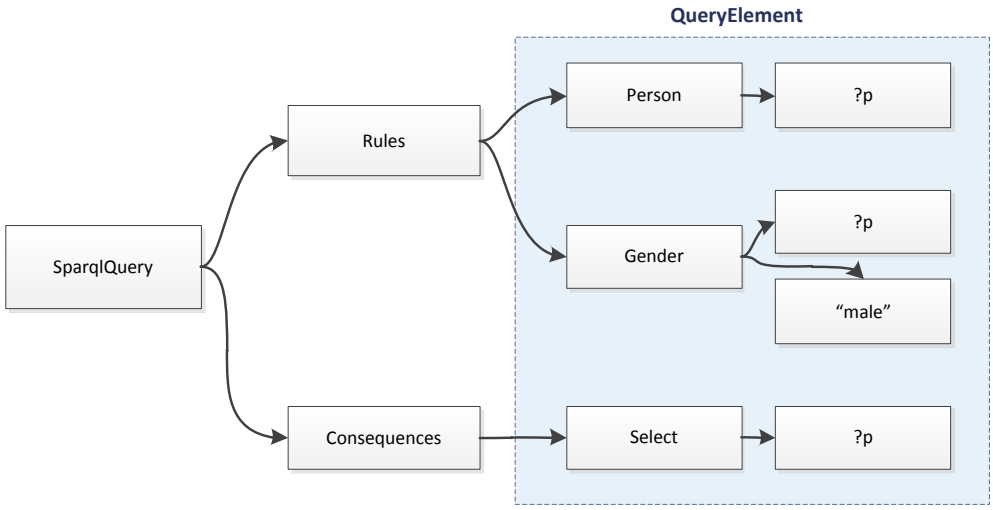

Figure 12 Structure of a Parsed SQWRL Query

A converter will convert the query object to Cascalog query using steps described in 3.3.1. The result is a CascalogQuery object derived from the Query class. Such object then can be used to generate Cascalog query source code which can be run by Clojure runtime.

4.5 Query Execution

The Query Execution component is responsible for executing queries and fetches the result to user's local machine. The component provides some interfaces so different query strategy can be added. Currently the query execution supports executing Cascalog source files on Hadoop and Amazon MapReduce.

4.5.1 Execution Steps

During a query execution, three steps will be performed in turn: deploy, executing, result fetching and data clear. It can be seen in the diagram followed.

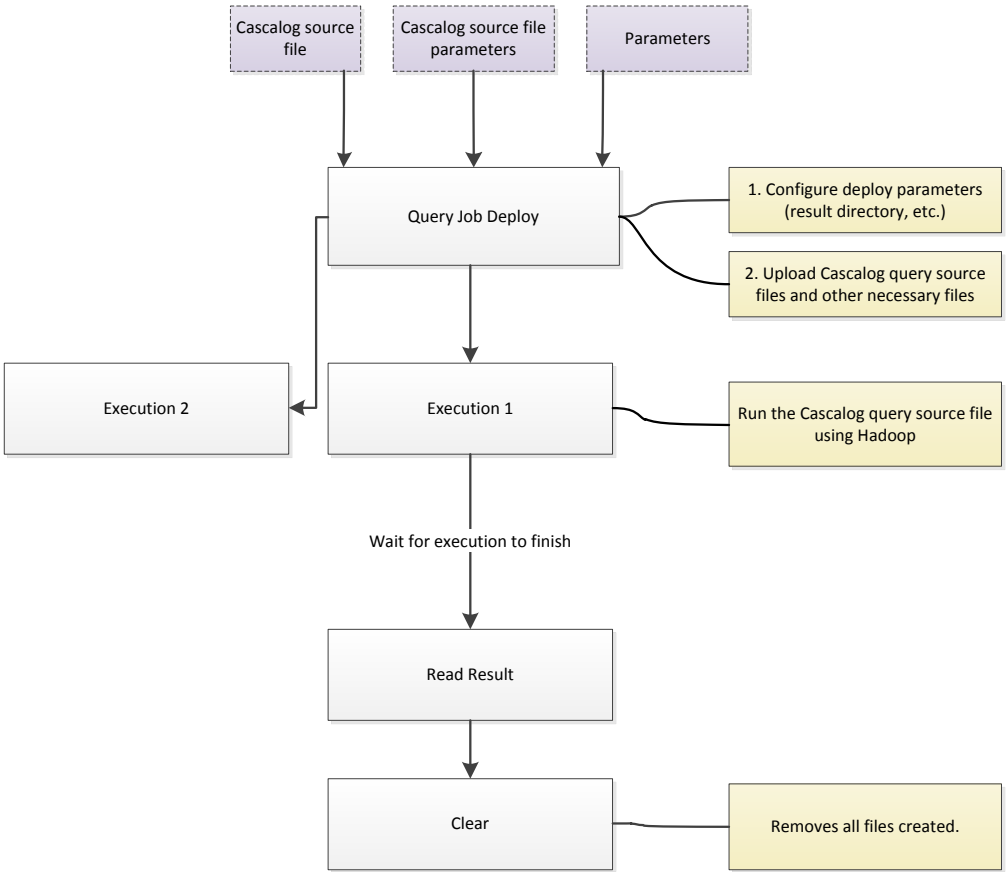

Figure 13 Query Execution Steps

First, the caller should have the Cascalog query source to execute. Callers also need to provide some parameters for execution, including:

- Data source directory or Triple Store base directory
- Output directory, where the output files are. When triple store is used, the output directory becomes the base directory of the Triple Store which is used to save the result.
- Deploy directory, where files required by query execution to be uploaded. Such files

including Cascalog source file and query parameter file.

However, with different strategy, parameters may be different. For example when using a triple store, one may also need to provide the name of the Triple Table which for query result output.

With all required execution information, the first step – **Deployment** is performed. Most deploy strategies require this step. Deployment is to upload all files required by execution to the execution environment (Hadoop). Each deployment will be assigned a unique ID, and files will be uploaded to a directory containing this ID in its name. This is to ease the file management, and also makes it easy to remove all deployed files from environment.

Each deployment can be **executed** multiple times. Like deployment, each **execution** also has a unique ID for the same reasons. **The Execution Step** does the following things: 1) creates and configures the MapReduce Job, 2) submits the job to Hadoop and run it and 3) gets job status and result if it is completed. Since we use a client-server model and the execution component runs on client side, it may wait for a job to complete synchronized or asynchronized. When running in asynchronized mode, user can check job status and retrieve result at any time.

The Clear Step clears all data created during the query execution. After this step, the environment should look like the query has never been executed. This step involves interaction with the Triple Store and the States Provider. Since the States Provider design is still immature, the Clear step is not actually performed.

4.5.2 Execution Strategies

Different execution approaches are needed to deploy and execute Cascalog queries on different environment. For each approach, it has to have a corresponding execution strategy in the system. An execution strategy consists of two key classes, QueryDeployment and QueryExecution:

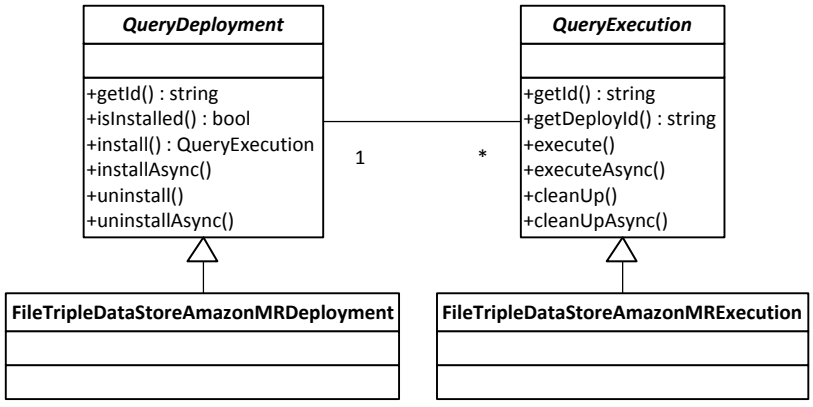

Figure 14 Base Classes for New Execution Strategy

- *QueryDeployment*. This class is the actor of the Deployment Step, which is responsible for configuring the target environment and uploading necessary files. Subclass should

implement the *install()* method to do the actual work, and returns a new QueryExecution for executing the query. *uninstall()* method is not used in this version.

- *QueryExecution*. This class is the actor of the Execution Step. It is responsible for executing query in target environment. Subclass should implement the *execute()* method to execute the query.

Currently, the system provides 4 strategies. Their supported environment is listed below:

Execution Environment	Data Source Provider
Amazon Elastic MapReduce	Plain text files on S3N
Amazon Elastic MapReduce	Triple Store on S3N
Local Hadoop Cluster	Plain text files on HDFS
Local Hadoop Cluster	Triple Store on HDFS

Table 7 Execution Strategies

Using these strategies, the system can run queries on Amazon Elastic MapReduce or a Hadoop Cluster, with either plain text files or Triple Store as data source and output. In this version, which strategy is used is still hard coded; however, using configuration files could be a better way to make switches.

4.6 A Sample Run

In this part we present how system would function when running a SWARQL query on Amazon MapReduce. Triple Store is used as data source and output destination. Related code will also be presented.

Following RDFS and RDF is used in the example:

```
<?xml version='1.0' encoding='UTF-8'?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:owl="http://www.w3.org/2002/07/owl#"
        xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

    <owl:Class rdf:ID="Student">
      <rdfs:label>Students</rdfs:label>
    </owl:Class>

    <Student rdf:about="http://tempuri.org/student1" />
    <Student rdf:about="http://tempuri.org/student2" />
    <Student rdf:about="http://tempuri.org/student3" />
    <Student rdf:about="http://tempuri.org/student4" />
    <Student rdf:about="http://tempuri.org/student5" />
</rdf:RDF>
```

4.6.1 Prepare the Data

To upload the RDF to system, we need to extract triples out of the ontology manually. Since the student class has no property, only triples contains relationship between the class name and the individual ID have been extracted:

```
Student | http://tempuri.org/student1
Student | http://tempuri.org/student2
Student | http://tempuri.org/student3
Student | http://tempuri.org/student4
Student | http://tempuri.org/student5
```

Save the last column to file “student.txt”, thus this file contains all individual IDs of class student.

4.6.2 Upload Data to System

Run the system jar from command line or terminal, use following command:

```
query.jar -e -a= student.txt
```

After the system started, it will parse the arguments and check if the preconfigured triple store exists:

```
if(!FileTriplebase.exists(triplebaseDir))
    base = FileTriplebase.create(triplebaseDir);
else
    base = new FileTriplebase(triplebaseDir, Access.ReadAndWrite);
```

The class *FileTriplebase* represents a Triple Store that saves data in files. The *create(string)* method is used to create a new Triple Store under target directory (only important statements are shown):

```
// creates directories
config.getDirectory().create();
config.getTablesDirectory().create();

// create a state bag to store internal states, FileTriplebaseStates is a helper class to access
// states in the states bag with get/set methods instead of state names.
IStatesBag bag = statesProvider.newBag(FileTriplebase.class, "triplebase", "triplebase");
FileTriplebaseStates states = new FileTriplebaseStates(bag);
states.setCreateTime(new Date());

// save the states
```

```
states.getStatesBag().applyChanges();
statesProvider.applyChanges();
```

The system will then check if a triple table with the same name as the file exists, if not, it will create one:

```
tableName = tableName.toLowerCase();

DirectoryInfo dir = this.getConfig().getTablesDirectory().getDirectory(tableName);
dir.create();

// create and initialize a states bag for the new table
IStatesBag states = this.getStatesBagProvider().newBag(TripleTable.class, null, statesBagName);
states.put("basedir", dir.getUri());
states.applyChanges();
this.getStatesBagProvider().applyChanges();

// add the table name to table name list
ArrayList<String> tables = (ArrayList<String>) this.states.get(StateName_Tables);
tables.add(tableName);
```

Now we are ready to add triples. First a Triple Row Reader is created to read triples from the student.txt file:

```
java.io.File gf = new java.io.File(directory.getAbsolutePath()+"\\"+filelist[i]);
FileInputStream gfs = new FileInputStream(gf);
InputStreamReader gisr = new InputStreamReader(gfs);
BufferedReader gbr = new BufferedReader(gisr);

DelimitedTextLineTripleRowReader tripleReader =
                                    new DelimitedTextLineTripleRowReader(gbr, " ");
```

Class DelimitedTextLineTripleRowReader sees each line as a triple, and separate a line into parts with a given separator for example a space or a “|”, and one part is considered as a triple element.

```
TripleRow row = null;
EncodedTripleRowWriter writer
                                    = new EncodedTripleRowWriter(this.dataStore.append(), encoder);
try {
      while((row = reader.read()) != null)
            writer.write(row);
}
finally
```

```
    writer.close();
```

In above code, a TripleRow is written to the table using an EncodedTripleRowWriter instance. EncodedTripleRowWriter takes a triple object -- which has elements in strings -- and convert each element to bytes using an encoder. After this, triples in bytes will be written to the underlying DataRowWriter:

```
byte[][] cells = new byte[row.getCells().length][];
for(int i = 0; i < cells.length; i++)
    cells[i] = this.encoder.encodeCell(row.getCells()[i]);
this.writer.writeRow(cells);
```

The DataRowWriter is obtained from the Data Store belongs to the table. According to configuration, a proper DataRowWriter implementation will be used. Normally we use the Info Leading Format, which is less effective but can store variable length row with variable size cells.

```
this.out.writeShort(DataRowStreams.ROW_STATE_Normal);
this.out.writeByte(cell.length);
for(int i = 0; i < cell.length; i++) {
    byte[] c = cell[i];
    this.out.writeShort(c.length);
    this.out.write(c);
}
```

The output stream used by DataRowWriter, is a SplitedFileOutputStream object, it manages file splits while writing bytes. Its write(int) method inherited from the OutputStream class is optimized so the splitting logic will not affect performance too much:

```
try {
    if(this.shouldSplit())
        nextSplit();
    position++;
    this.currentOut.write(b);
    if(this.splitListener != null)
        this.splitListener.bytesWritten(1);
}
catch (Exception ex)
    throw new IOException(ex);
```

The split listener used in above codes monitors data writing in a split, and tracks data size and row counts in each split. When current split is closed, it applies tracked information to DataStore.

```
@Override
public void rowWritten(int cellCount) {
```

```
        this.splitMgr.updateSplitInfo(split.getIndex(), ++currentSplitRowCount);
        this.dataStore.increaseRowCount(1);
}

@Override
public void splitClosed() {
        split.setFileEndPosition(this.currentSplitPosition);
        this.splitMgr.updateSplitFilePosition(split.getIndex(), split.getFileEndPosition());
        this.splitMgr.applyToStates();
        this.split = null;
}
```

When all triples have been written, writers will be closed. Upon the close, each component has to apply their internal states to their states bag.

4.6.3 Execute a Query

Before a query can be executed on Amazon MapReduce, the application package must be uploaded to Amazon S3N. In this sample, the uploaded file path is s3n://sample/executor.jar.

Same as previous, a user needs to issue a command line/terminal command to run the application:

```
query.jar -q /home/roy/Desktop/query.txt -o=/home/roy/Desktop/query_output.txt -s=/home/roy/Desktop/query_stat.txt
```

Above command indicates the queries need to run can be found in file "/home/roy/Desktop/query.txt", result and query statistics will be saved to "/home/roy/Desktop/query_output.txt" and "/home/roy/Desktop/query_stat.txt" respectively. All queries in the query file will be executed in turn. Assume we have only one query in it:

```
Student(?p) -> sqwrl:select(?p)
```

It is a simple SWARQL query that queries all student names. After getting the query, the system will first try to rewrite it using a reasoner. This part is not in the scope of this thesis so its implementation details are not shown here. The output of this phase is a collection of SPARQL queries in strings:

```
List<String> sqwrls = queryFormation(rewrite, query);
```

These queries will then be converted to CascalogQuery objects using the QueryConversion component. Since the whole process is complex, only high level calls are shown below:

```
GeneralQueryConvertor conventor = new GeneralQueryConvertor();
```

```
ArrayList<CascalogQuery> cq = new ArrayList<CascalogQuery>();

for (String query : queries) {
        cq.add((CascalogQuery)conventor.convert(SqwrlQuery.parse(query), "cascalog"));
}
```

Then a TripleDataTapOnAmazonMapReduce object is used to generate Cascalog query (actually, it is Clojure source code) that can be run in Amazon MapReduce.

```
TripleDataTapOnAmazonMapReduce context = new TripleDataTapOnAmazonMapReduce();
context.setNamespace("test");
context.setUseCounter(true);
context.setUseLogger(true);
CommonCodeWriter codeWriter = new CommonCodeWriter();
ClojureSourceInfo sourceInfo = context.writeCodeInContext(codeWriter, cq);
String contextQuery = codeWriter.getCode();
```

Now we can upload the source code to Amazon S3:

```
StorageFactory.getFile(targetFileUri).updateOrCreateWith(
                new StringBufferInputStream(contextQuery), true);
```

By now, the query is deployed on Amazon S3. To run the query, we need to create a MapReduce job:

```
RunJobFlowRequest runJobFlowRequest = new RunJobFlowRequest();
runJobFlowRequest.setName("execution_" + this.getId());

HadoopJarStepConfig hadoopJarStep = new HadoopJarStepConfig();
hadoopJarStep.withJar(this.clojureSourceFileUri)
                                        .withMainClass(this.executorClassName)
                                        .withArgs(executorParams.toArgs());
StepConfig config = new StepConfig("Custom Jar", hadoopJarStep);
runJobFlowRequest.getSteps().add(config);

// configures the nodes
JobFlowInstancesConfig instanceConfig = new JobFlowInstancesConfig();
instanceConfig.setInstanceCount(10);
instanceConfig.setHadoopVersion("0.20");
instanceConfig.setMasterInstanceType("m1.large");
instanceConfig.setSlaveInstanceType("m1.large");
runJobFlowRequest.setLogUri("s3n://sample/debug/");
runJobFlowRequest.setInstances(instanceConfig);
```

Code above creates a MapReduce job on Amazon MapReduce, using 10 large nodes and Hadoop 2.0. The job is configured to run the ClojureExecutor, which will then read the uploaded Cascalog query file and execute it in Hadoop.

During the execution, job status is checked time after time. Following code is used to query job status:

```
DescribeJobFlowsRequest request = new DescribeJobFlowsRequest();
request.withJobFlowIds(this.runJobFlowResult.getJobFlowId());
DescribeJobFlowsResult response = this.service.describeJobFlows(request);
return response.getJobFlows().get(0).getExecutionStatusDetail();
```

Once the job is completed, result will be fetched from the output Triple Store on Amazon S3:

```
FileTriplebase tb = new FileTriplebase(databaseDir, Access.Read);
tb.initialize();
TripleRowReader reader = tb.getTable(tableName).read();
TripleRow row = null;
while((row = reader.read()) != null)
    fileStreamWriter.write(row.toString());
```

4.6.4 What Happened In Hadoop?

The MapReduce job submitted guides Hadoop to run a static main method in class ClojureExecutor. The main() method reads Cascalog query code into memory and executes it using Clojure runtime:

```
RT.init();
String clojureStr = clojureFile.readAllText();
clojure.lang.Compiler.load(new StringReader(clojureStr));
Var mainFunc = RT.var(
        para.getClojureSourceInfo().getNamespace(),
        para.getClojureSourceInfo().getQueryFuncName());
mainFunc.invoke(para.getInvokeParameters());
```

At this point, the Cascalog starts doing its work. It parses the query into Cascading Flows, which generates Hadoop MapReduce jobs that does the actual work. The Cascalog query looks like:

```
(ns test (:use cascalog.api) (:require [cascalog [vars :as v] [workflow :as w] [ops :as c]])
(:gen-class))

(defn sink_tap [sink_datastoreDir sink_tableName ]
    (let [scheme (DataStoreTapScheme. )]
        (DataStoreTap. sink_datastoreDir sink_tableName scheme )))
```

```
; query function
(defn source_tap_Person [source_Person_datastoreDir]
    (let [scheme (DataStoreTapScheme.)]
        (DataStoreTap. source_Person_datastoreDir "Person" scheme )))

(defn source_tap_Gender [source_Gender_datastoreDir]
    (let [scheme (DataStoreTapScheme.)]
        (DataStoreTap. source_Gender_datastoreDir "Gender" scheme )))

(defn query_1 [source_Person_datastoreDir source_Gender_datastoreDir sink_datastoreDir
sink_tableName]
    (let [Person (source_tap_Person source_Person_datastoreDir)
            Gender (source_tap_Gender source_Gender_datastoreDir)
            sink (sink_tap sink_datastoreDir sink_tableName)]
        (?<- sink [?p ](Person ?p )(Gender ?p (. Gender encodeCell "male" )))))

(defn -main [args]
    (let [sink_datastoreDir (get args 0)
        sink_tableName (get args 1)
        source_Person_datastoreDir (get args 2)
        source_Gender_datastoreDir (get args 3)]
    (query_1 source_Person_datastoreDir source_Gender_datastoreDir sink_datastoreDir
sink_tableName)))
```

The query uses Triple Store Taps as both source and sink. The Triple Store Tap implements Hadoop's IO interfaces to read and write data from and to a Triple Store. It sets DataRowInputFormat and DataRowOutputFormat as Hadoop's Input and Output format, so Hadoop knows how to access data with it. We access triples at Data Store level. The reason is triple bytes can be accessed directly from a Data Store, which avoids overhead of decoding triple bytes to strings. Following statements show how Hadoop reads data from a Triple Table:

```
if(!initialized) {
    FileTriplebase tbase = new FileTriplebase(tSplit.getTriplebaseDir(), Access.Read);
    tbase.initialize();
    ds = (FileDataStore) tbase.getTable(tSplit.getTableName()).getDataStore();
    reader = ds.read();
}

if(reader.nextRow()) {
    key.set(this.reader.getCurrentRowIndex());
    ArrayList<UnsignedVariableIntegerWritable> cells = new
ArrayList<UnsignedVariableIntegerWritable>();
    UnsignedVariableInteger c = null;
```

```
        while ((c = reader.readCellsAsInteger()) != null) {
                cells.add(new UnsignedVariableIntegerWritable(c));
        }

        value.set(cells.toArray(new UnsignedVariableIntegerWritable[cells.size()]));
}
```

First a Data Store Reader will be created for reading. Then each time Hadoop asks for a new value, it read cells bytes as integer arrays and saves them using an UnsignedVariableIntegerWritable array, which works with Hadoop's object serialization mechanism. Each converted array will be regarded as a Tuple and operated by Cascading, and the result Tuples will be send to Hadoop output. The output process is similar to input:

```
Writable[] values = value.get();
UnsignedVariableInteger[] cells = new UnsignedVariableInteger[values.length];
for(int i = 0; i < cells.length; i++) {
        UnsignedVariableIntegerWritable cell = (UnsignedVariableIntegerWritable)values[i];
        cells[i] = cell.get();
}
writer.writeRowAsInteger(cells);
```

5 Tests

Our tests aim to find the performance of the system and the aspects affect the performance. We concluded the two aspects that may have performance hit: the cluster configuration and data encoding. Tests are grouped into the two categories.

5.1 Test Environment

We use Amazon MapReduce service to test the system performance. The Amazon MapReduce service is a distributed service that helps users to run their Hadoop MapReduce jobs without configuring Hadoop nodes themselves. Users can configure the number of nodes they want to use, and the nodes' hardware configuration, and the service will allocate virtual machines and configure Hadoop to run the job. It is flexible for users that need to run lots of jobs with different job configuration. Also to make all tests under same condition so they could be comparable, all the tests have been run using the Amazon MapReduce service.

In our tests, two node configurations have been used:

- **Small Instance**. 1.7 GB of memory, 1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit), 160 GB of local instance storage, 32-bit platform [11].
- **Large Instance**. 7.5 GB of memory, 4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each), 850 GB of local instance storage, 64-bit platform [11].

5.2 Test Data and Queries

The Lehigh University Benchmark (LUBM) provides queries and ontology for Semantic Web repositories evaluation. It also provides a data generator to generate test data from the given ontology. LUBM is often used in performance and correctness tests on repositories. We used the ontology and queries provide by LUBM in tests.

The ontology from LUBM describes university domain knowledge. The data generator from LUBM can produce randomly generated university information in OWL files. A small piece of python program is used to convert OWL data to triples and save them in plain text files.

LUBM defines 14 queries for tests; however, since most of those queries involve the use of a reasoner to cover the whole data set, which is out of the scope of this thesis, only following queries are used:

- **Query 1.** It queries graduate students who take a specific course. "This query bears large input and high selectivity. It queries about just one class and one property and does not assume any hierarchy information or inference. [12]" The SPARQL version of the query is:

```
GraduateStudent(?0)                                                    ^
takesCourse(?0,<http://www.Department0.University0.edu/GraduateCourse0>)   ->
```

sqwrl:select(?0)

- **Query 2**. It queries graduate students as well as their university and department name; students should have undergraduate degrees from the same university as the one where they got or will get their graduate degree. “This query increases in complexity: 3 classes and 3 properties are involved. Additionally, there is a triangular pattern of relationships between the objects involved. [12]” The SPARQL version of the query is:
  GraduateStudent(?X) ^ University(?Y) ^ Department(?Z) ^ memberOf(?X,?Z) ^ subOrganizationOf(?Z,?Y) ^ undergraduateDegreeFrom(?X,?Y) -> sqwrl:select(?X,?Y,?Z)

- **Query 14**. It queries all undergraduate students. “This query is the simplest in the test set. This query represents those with large input and low selectivity and does not assume any hierarchy information or inference. [12]” The SPARQL version of the query is:
  UndergraduateStudent(?X) -> sqwrl:select(?X)

5.3 Performance Dependent on Cluster Configuration

Cluster configuration is how we setup the Hadoop nodes, for example the number of Hadoop nodes, the processor frequency of node machines, and the number of reducers. These configurations are factors that outside the system’s internal logic, but could have effect on its performance.

Number of Nodes is how many nodes are used to run a Hadoop MapReduce Job. Odd numbers between 2 to 20 are tested.

Number of Reducers is how many nodes are used as reducers. Assuming the number of nodes is n, 1, (n / 2), and (n – 1) reducers are tested.

Node Configuration, as described in 5.1, it is the hardware and network configuration of nodes. See 5.1 for the two configurations used in tests.

Number of Universities. 50, 100, 200, 1000 universities data are used respectively in our tests.

Plain text files are used as data source. Plain texts are stored on Amazon S3 service and will be copied to Hadoop cluster’s HDFS before query execution starts to avoid overhead of Amazon S3 IO [Tomasz [13]]. Following test results have been published in paper “Performance Analysis of Hadoop for Query Processing [13]” at WAINA 2011.

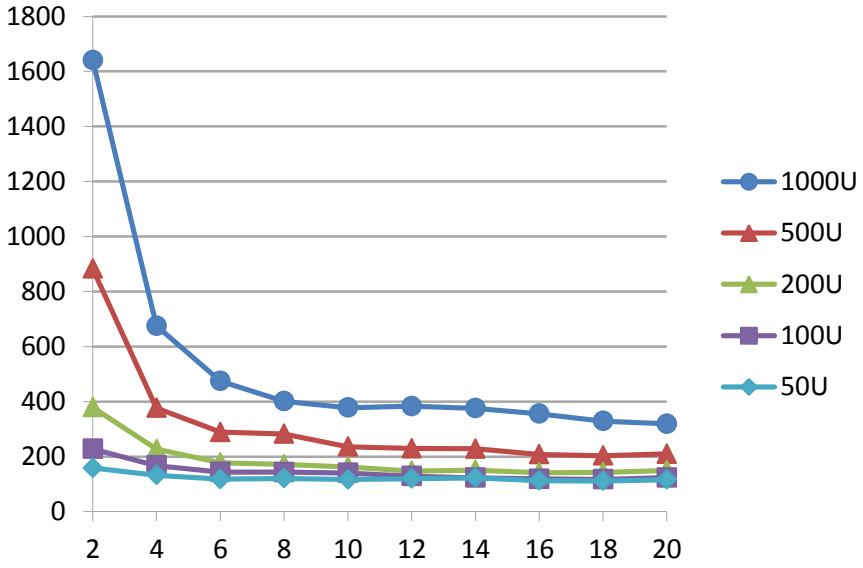

Figure 15 Time (seconds) cost by executing Query 1 with n nodes (small instances, 1 reducer)

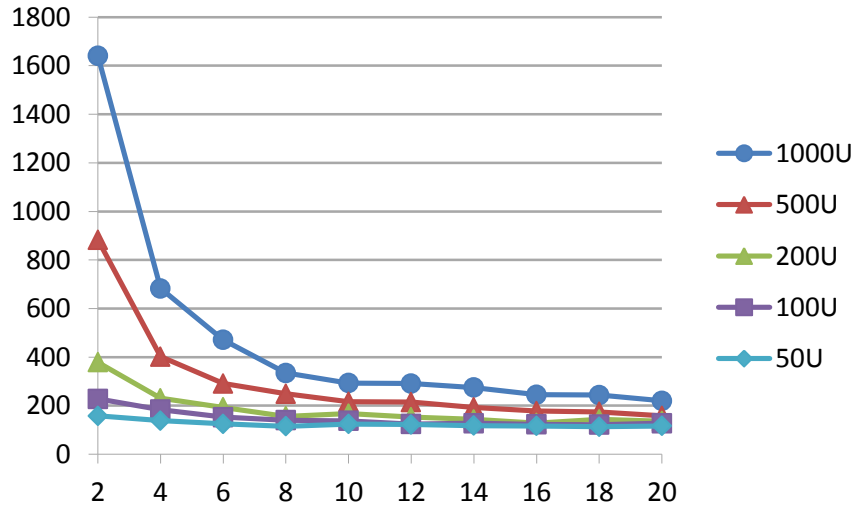

Figure 16 Time (seconds) cost by executing Query 1 with n nodes (small instances, n / 2 reducers)

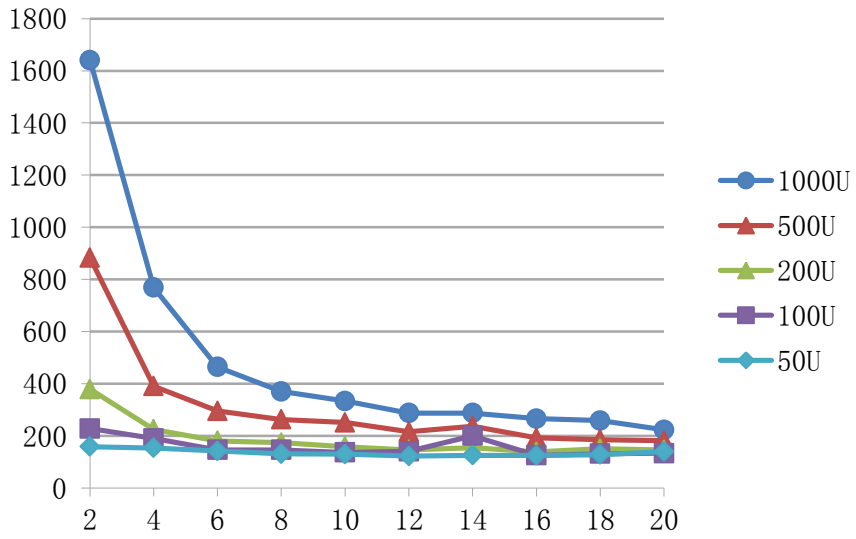

Figure 17 Time (seconds) cost by executing Query 1 with n nodes (small instances, n - 1 reducers)

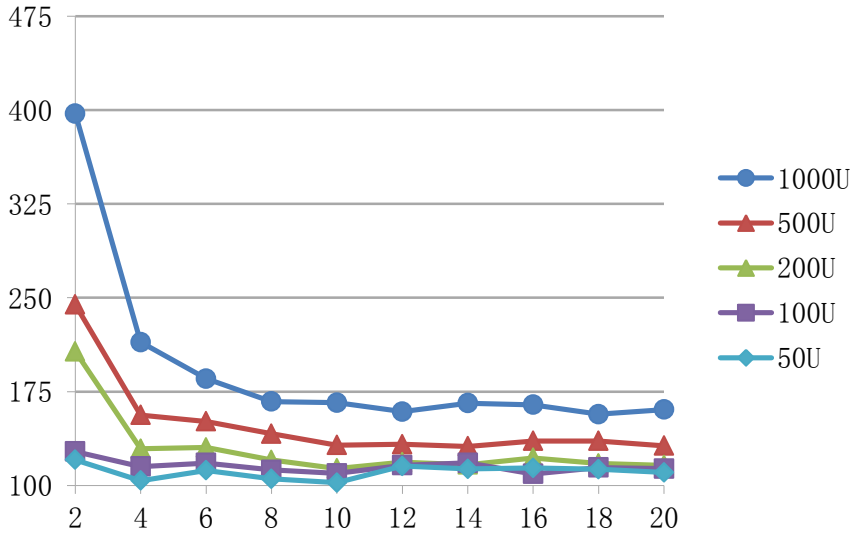

Figure 18 Time (seconds) cost by executing Query 1 with n nodes (large instances, 1 reducer)

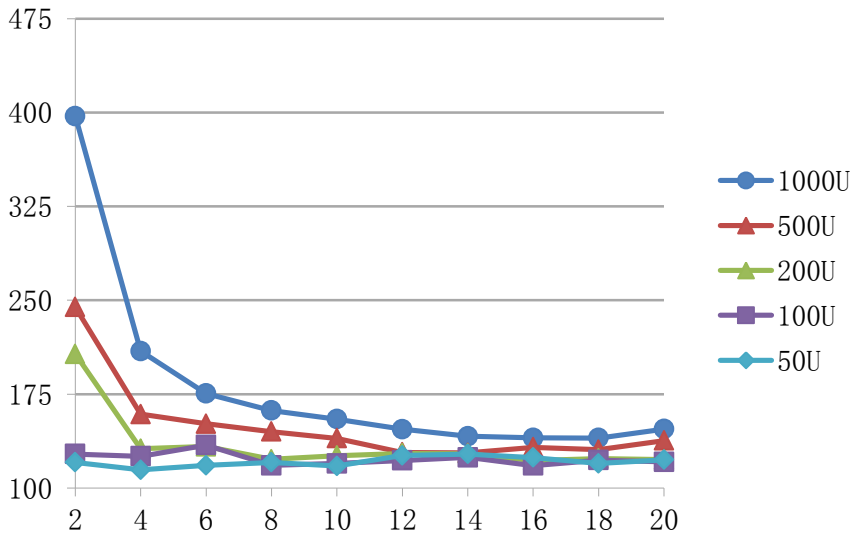

Figure 19 Time (seconds) cost by executing Query 1 with n nodes (large instances, n / 2 reducers)

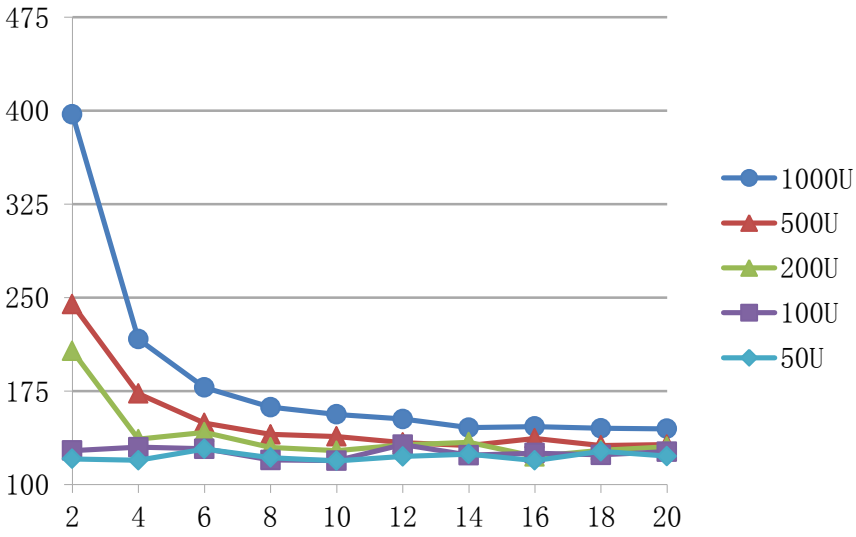

Figure 20 Time (seconds) cost by executing Query 1 with n nodes (large instances, n - 1 reducers)

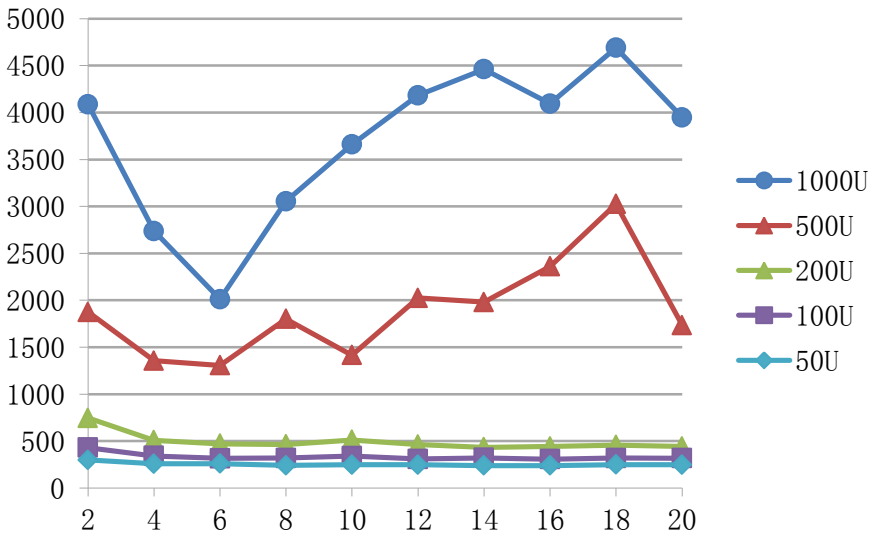

Figure 21 Time (seconds) cost by executing Query 2 with n nodes (small instances, 1 reducer)

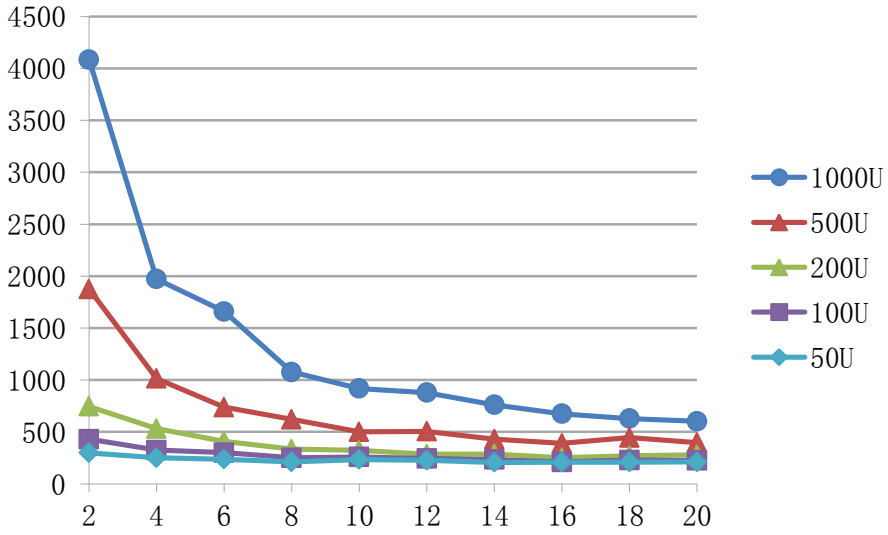

Figure 22 Time (seconds) cost by executing Query 2 with n nodes (small instances, n / 2 reducers)

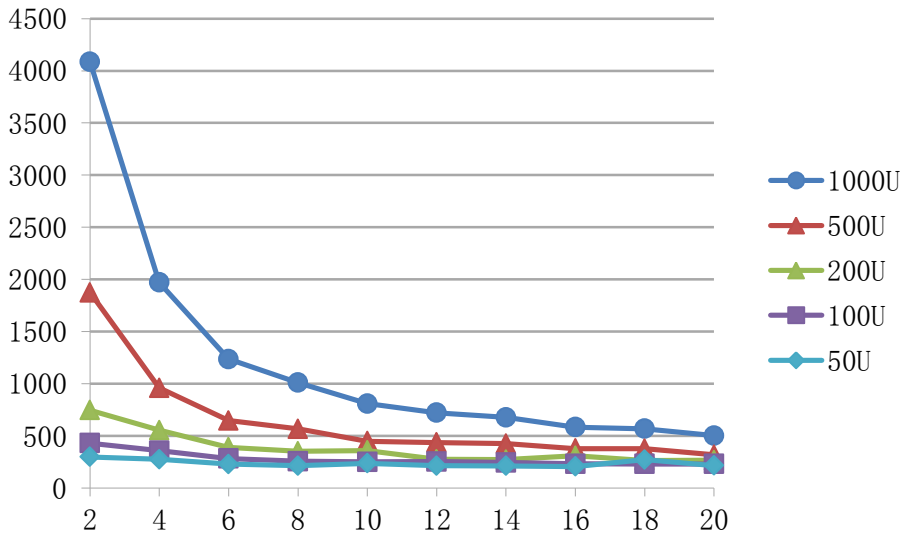

Figure 23 Time (seconds) cost by executing Query 2 with n nodes (small instances, n - 1 reducers)

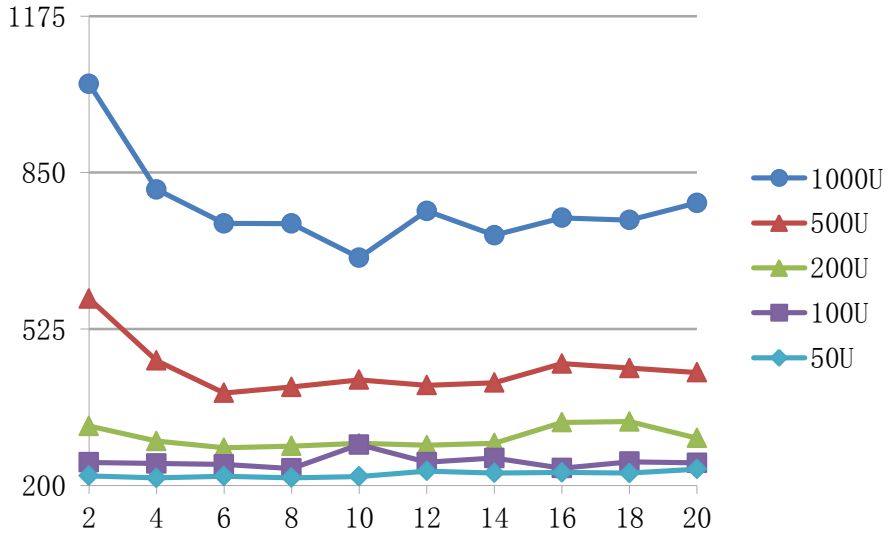

Figure 24 Time (seconds) cost by executing Query 2 with n nodes (large instances, 1 reducer)

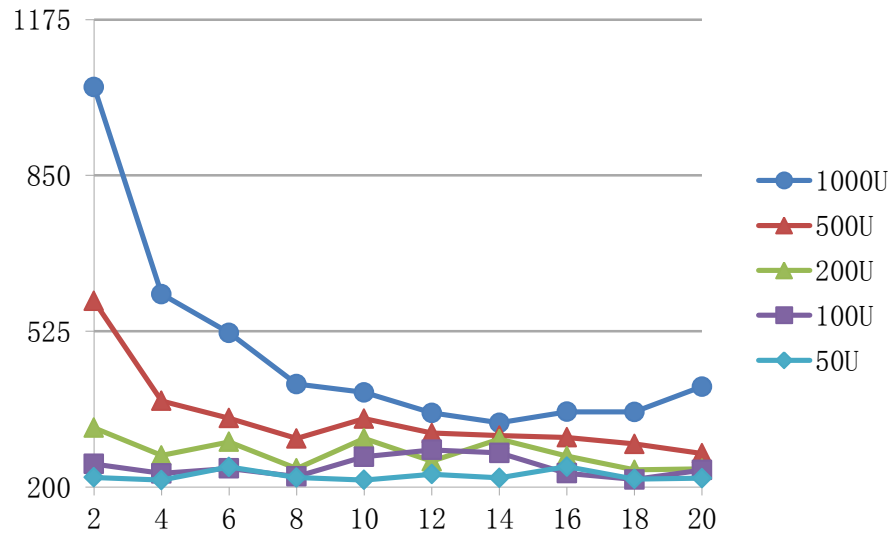

Figure 25 Time (seconds) cost by executing Query 2 with n nodes (large instances, n / 2 reducers)

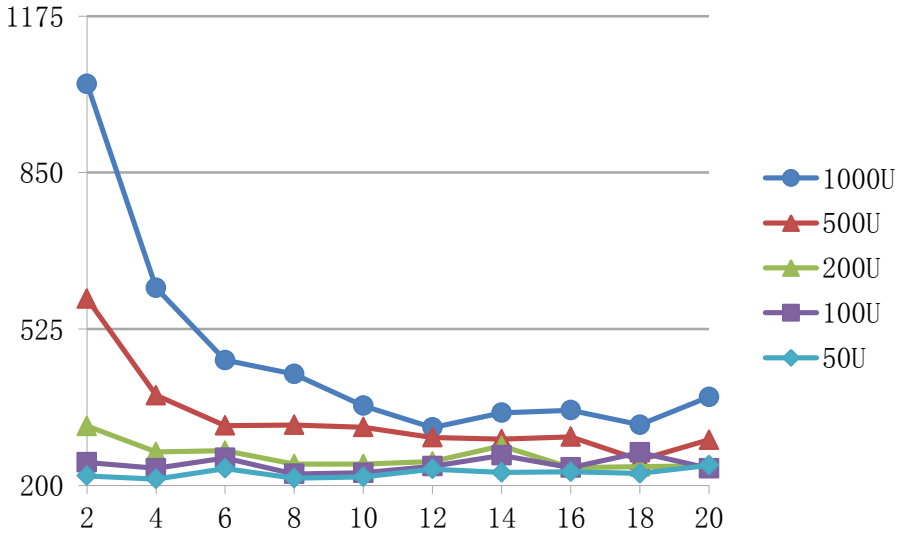

Figure 26 Time (seconds) cost by executing Query 2 with n nodes (large instances, x -1 reducers)

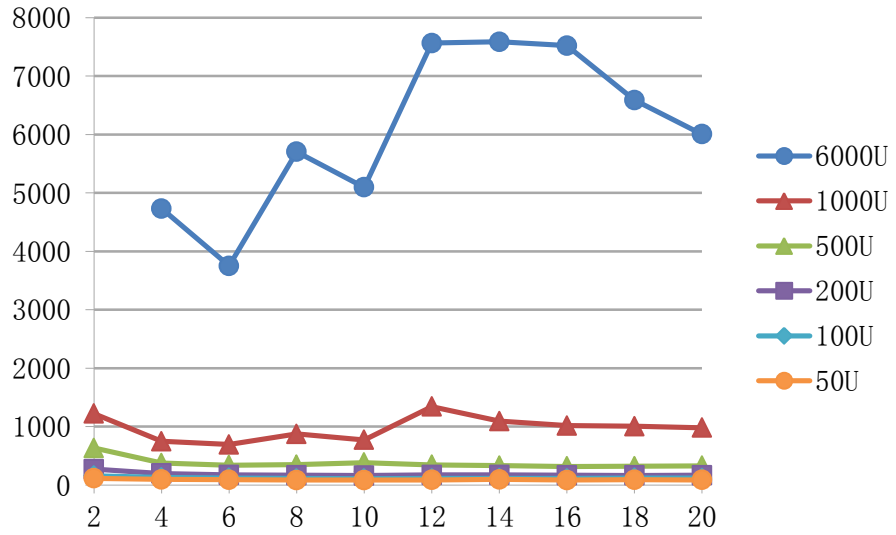

Figure 27 Time (seconds) cost by executing Query 14 with n nodes (small instances, 1 reducer)

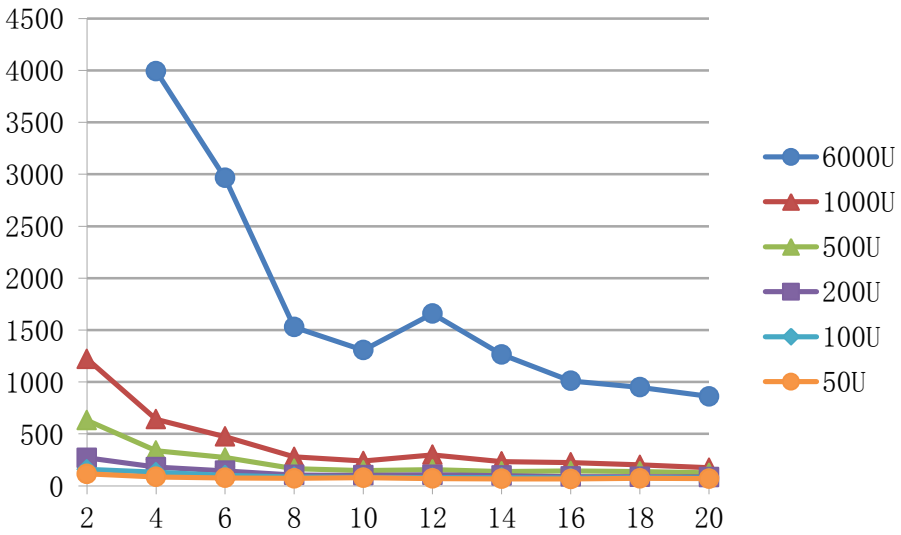

Figure 28 Time (seconds) cost by executing Query 14 with n nodes (small instances, n / 2 reducers)

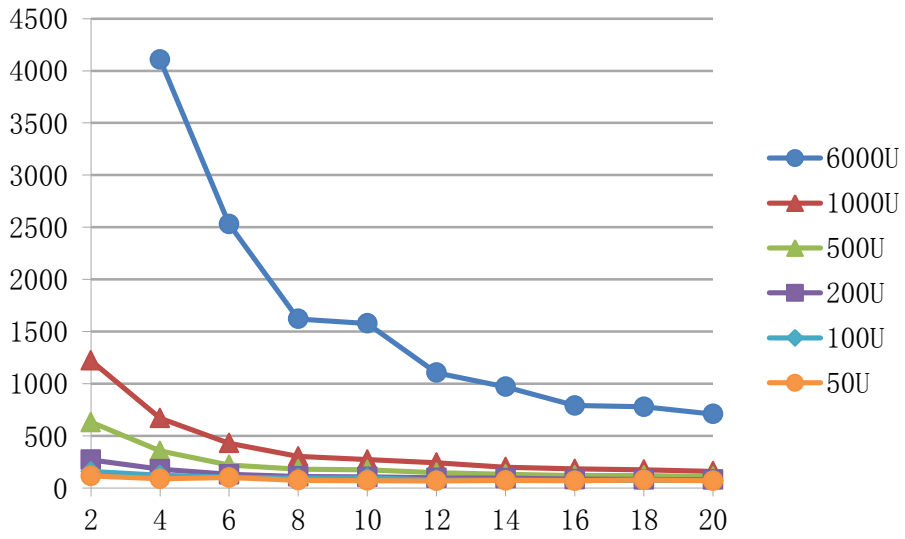

Figure 29 Time (seconds) cost by executing Query 14 with n nodes (small instances, n - 1 reducers)

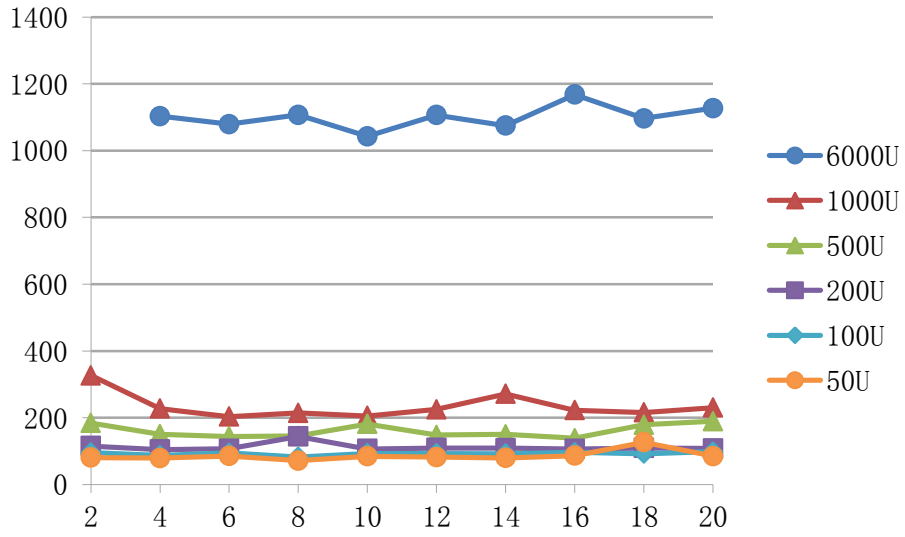

Figure 30 Time (seconds) cost by executing Query 14 with n nodes (large instances, 1 reducer)

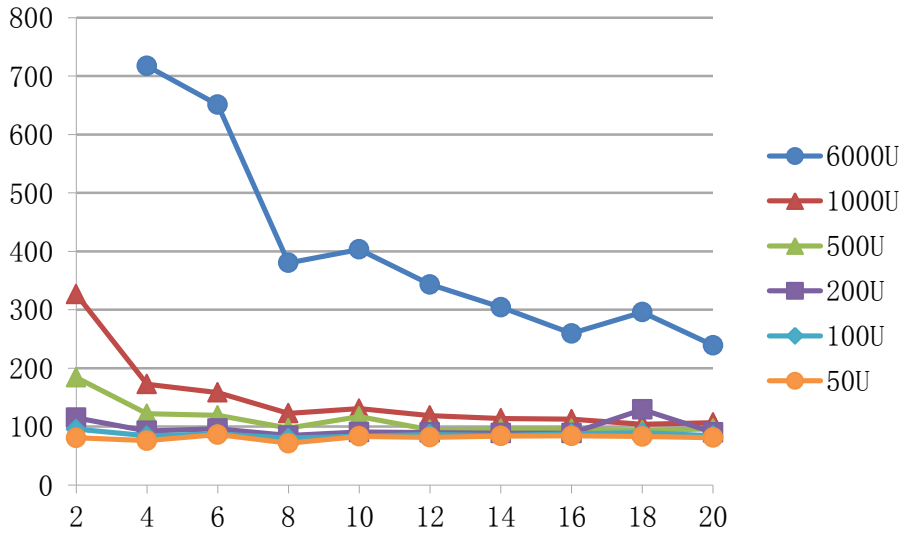

Figure 31 Time (seconds) cost by executing Query 14 with n nodes (large instances, n / 2 reducers)

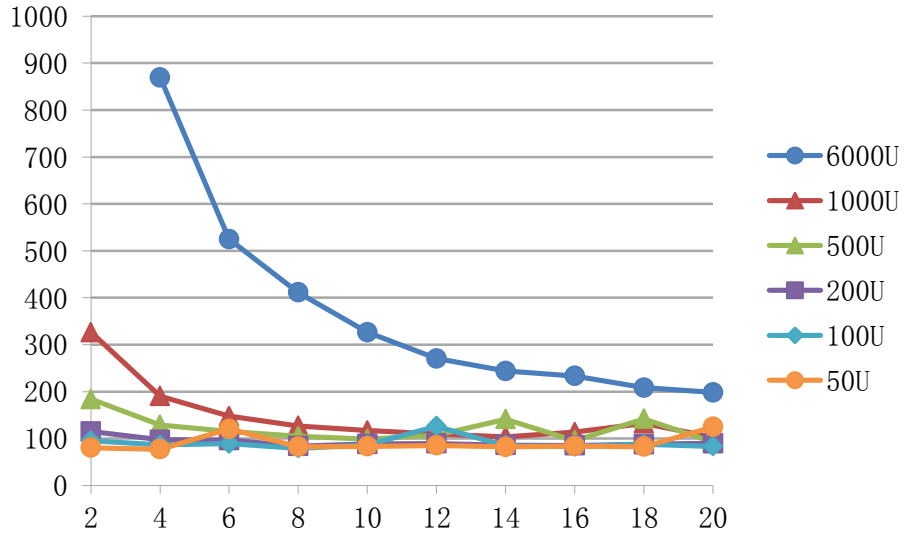

Figure 32 Time (seconds) cost by executing Query 14 with n nodes (large instances, n - 1 reducers)

5.4 Performance Dependent on Data Structure

This section has been submitted as part of the paper "Evaluation of Some Optimization Techniques for Semantic Query Answering on Shared-nothing Architecture [16]" submitted to AINA - 2011 Special Issue of International Journal of Space - Based and Situated Computing (IJSSC).

Two data encodings have been prebuilt in the system: plain text and variable integer encoding. For integer encoding, two data structures can be used to hold values in Hadoop: byte array and integer array. The details of these data encoding and structures can be found in 4.3.3 Data File

Encoding. Tests in this part are carried out with following data structure and encoding sets:

Encoding	Data Structure
Plain Text	String
Integer Encoding	Byte array
Integer Encoding	Integer array

Data files are saved on Amazon S3 and will be read directly by Hadoop. Size of data used in each query with different encoding are listed below:

	Plain Text	Integer Encoding	Involved Triple Element Count
Q1	1666 MB	363 MB	15664758
Q2	749 MB	201 MB	15291035
Q14	248 MB	48 MB	3961133
Mapping Dictionary Data Size	N/A	41 MB	

Table 8 Sizes of Data and Triple Element Count Used in Queries

From Table 8 we can see, integer encoded data is nearly 5 times smaller than plain text files. When running queries with integer encoded data files, the mapping dictionary data may also need to be read, which is 41 MB for all queries. The total size of encoded data and mapping dictionary data is still smaller than plain text data is because the mapping dictionary does not contain duplicated strings.

	Object.hashCode()	Object.compareTo()	Writable.readFields()	Writable.write()
Q1	761	67992946	7561221	676
Q2	52820784	294343917	842790851	65882684
Q14	2621440	151591435	100182067	2621436

Table 9 Key Method Calls in Queries

Table 9 shows Cascading makes a large number of calls on the four basic methods. Object.hashCode() and Object.compareTo() are used by Cascalog and Hadoop to identify and compare values. Writable.readFiles() and Writable.write() are used by Hadoop to serialize and deserialize values. Each triple element object no matter we use string, byte array or integer array, has these four methods. This call number may be different in each run but does not fluctuate too much. Since time spent on a single method call is smaller than 1ms, we did not manage to track the total time spent on each method, but their affect can be seen from test result.

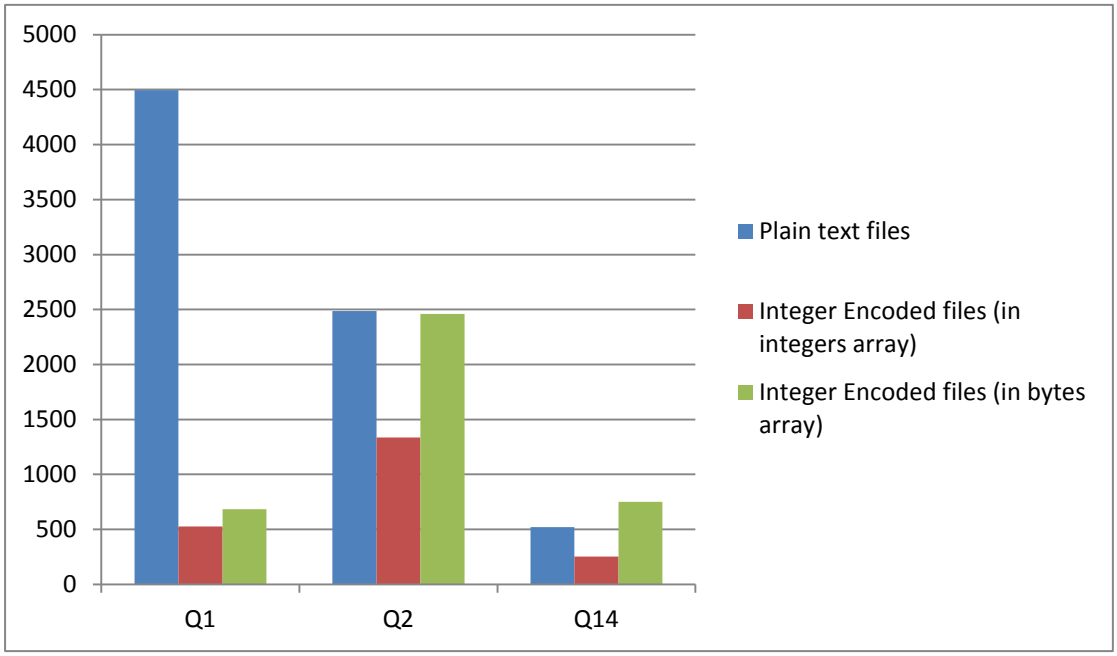

Figure 33 Execution Time in Seconds with Different Encoding and Data Structure for Q1, Q2 and Q14

Figure 33 shows execution time for Q1, Q2 and Q14 with different encoding. From it we can see when using plain text files, the query performance is the worst. On the other hand, integer encoded files with integer array data structure executes most efficiency in all queries.

A clear relationship between data size and execution time can be seen in tests using plain text files. With integer encoding, the impact of different data size on performance is not clear. For example, the time cost in Q1, which has to read the largest data set, is similar to Q14, which has the smallest data set.

This may somehow because of the complexity of queries. Among the three queries, their complexity from high to low is Q2, Q1 and Q14. Since Q14 has only one data source and no join; Q1 has two data sources and has to filter values, and Q2 has many more sources and needs at least 6 inner joins. Such complexity can also be seen from the number of generated MapReduce jobs as listed below:

Query	Number of MapReduce Jobs
Q1	1
Q2	5
Q14	1

Table 10 Number of Generated MapReduce Jobs for Queries

For tests using integer encoding, the results with different data structure also have big difference. From Figure 33 we can see, integer encoding with integer array performs much better than byte array. For example in Query 2, time used with byte array is near 2500 s, while with integer array, it is 1400 s, which is 1.8 times smaller than the other. The only difference with the two set of

tests are the data structure, and the cause of performance difference is the method calls on the value object. Table 9 shows the 4 methods which have been called most frequently by Hadoop and Cascading. We also have a small test on Object.hashCode() and Object.compareTo() performance on a local machine. The table below shows the average time (from 5 runs) used after calling the two methods respectively 500000 times on java.lang.String (type used to save strings in Java, provided by Java library), org.apache.hadoop.io**.**BytesWritable (class implements the byte array structure, provided by Hadoop library) and UnsignedVariableIntegerWritable (class implements the integer array structure, provided by us), execution environment is Windows 7 on a 2.2GHz CPU and 4 GB RAM machine.

	java.lang.**String**	org.apache.hadoop.io.**BytesWritable**	**UnsignedVariableIntegerWritable**
Object.hashCode()	84 ms	28 ms	10 ms
Object.compareTo()	81 ms	34 ms	10 ms

Table 11 Performance of Object.hashCode() and Object.compareTo() on a single machine

It can be seen that class used for byte array structure performs 3 times slower than the one for integer array. The reason is, the hashCode() and compareTo() methods need to iterate all elements in the array. Since a byte groups every 8 bits, and an integer groups every 32 bits, fewer steps are needed for iterating in an integer array since it has fewer elements.

6 Conclusion

In this thesis, we have described our approach to design and implement a semantic query execution system using Hadoop and Cascalog.

Both SPARQL and SQWRL are supported in this thesis. Internally, they are converted to Cascalog query source and submitted to Hadoop as a MapReduce job to be executed. On file system wise, HDFS, Amazon S3 and Java local file system are supported.

Two data encodings have been implemented in the system. Plain text encoding reads and writes triples to files using UTF-8 encoding, values read in Hadoop are saved in strings. Integer encoding substitute strings with integers, and stores the original strings in a string-integer mappings dictionary for reference. The dictionary groups items with string values' hash codes, and each group can be loaded into memory on-demand. Two data structure provided in the system for holding values read from integer encoded files. Byte array structure saves bits as a byte array, while integer array structure saves every 32 bits as an integer in an integer array.

Performance with plain text encoding, integer encoding with byte array and integer encoding with integer array have been tested and compared. The results show data size, query complexity and key methods efficiency affect query performance in different ways. For large data set, the overhead of read data is more than that of the other two aspects. While using smaller data files, performance affected by the overhead of large number of MapReduce jobs introduced by high query complexity can be seen clearly. When using same data encoding with same query, efficiency of the four basic methods can have great impact on performance. To sum up, integer encoding with integer array data structure wins the other two encoding and data structure set.

Reference

[1] Patrick van Bergen. 2009. Semantic web marvels in a relational database. http://techblog.procurios.nl/k/n618/news/view/34300/14863/Semantic-web-marvels-in-a-relational-database---part-I-Case-Study.html

[2] Ching-Long Yeh and Ruei-Feng Lin. Design and Implementation of an RDF Triple Store. 2002. http://datf.iis.sinica.edu.tw/Papers/2002datfpapers/sessionB/B-3.pdf

[3] Jena2 Database Homepage. http://jena.sourceforge.net/DB/

[4] Sesame Homepage. http://www.openrdf.org

[5] 4 Store Homepage. http://4store.org/

[6] http://en.wikipedia.org/wiki/Distributed_computing

[7] http://en.wikipedia.org/wiki/Semantic_Web

[8] http://en.wikipedia.org/wiki/Ontology_(computer_science)

[9] http://en.wikipedia.org/wiki/MapReduce#Uses

[10] http://www.cascading.org/1.2/userguide/html/ch03.html#N20110

[11] http://aws.amazon.com/ec2/#instance

[12] http://swat.cse.lehigh.edu/projects/lubm/query.htm

[13] Tomasz Wiktor Wlodarczyk, Yi Han, Chunming Rong, “Performance Analysis of Hadoop for Query Processing”, 2010

[14] Yi Han, Tomasz Wiktor Wlodarczyk, Yu Xiao, Chunming Rong, “Evaluation of Some Optimization Techniques for Semantic Query Answering on Shared-nothing Architecture”, 2011

[15] http://en.wikipedia.org/wiki/Ontology

[16] Y. Han, T. W. Wlodarczyk, X. Yu, C. Rong, “Evaluation of Some Optimization Techniques for Semantic Query Answering on Shared-nothing Architecture” submitted to AINA‐2011 Special Issue of International Journal of Space‐Based and Situated Computing (IJSSC)