

ABSTRACT

SRIDHAR, RADHIKA. Scaling Complex Analytical Processing on Graph Structured Data Using Map Reduce. (Under the direction of Dr Kemafor Anyanwu.)

Efficient analytical processing at the Web scale has become an important requirement as more decision support applications rely on the data on the Web. One approach for achieving the significant scalability is by the use of parallel processing techniques on a computational cluster of the commodity grade machines. Software platforms such as Map-Reduce, Hadoop and Pig are now available that allow the users to encode their tasks in terms of simple low-level primitives that are easily parallelizable. Further, a high-level dataflow language called the Pig Latin has been proposed for specifying analytical processing tasks using a mixture of the procedural and the declarative paradigms. This approach strikes a good balance between customizability and the potential for an automatic query optimization. However, the analytical processing capability currently offered by these frameworks is fairly basic and as such has narrow applicability to many real world scenarios. Furthermore, an increasing amount of data being made available on the Web is semi-structured. For example, some search engines report that the recent W3C standard for representing the metadata on the Web called the Resource Description Framework (RDF) already accounts for about 8,502,794 Web data URL's and 2,759,040

documents. However, such data is typically organized as a set of binary relations (a graph) whereas these frameworks are primarily targeted at processing the data structured as n-ary relational tables.

This thesis addresses the problem of enabling the scalable analytical data processing on the RDF datasets. Its approach is based on extending Yahoo's Pig system (an open source parallel processing) with constructs that allow complex data processing problems on the graph structured data to be expressed in a manner that is more amenable to automatic parallelization. Specifically, it makes the following contributions:

1. Extends Pig Latin, the dataflow language for Pig, with primitives that support the expression of queries in terms of a readily parallelizable multidimensional join operator, as well as support the expression of graph navigational filter expressions.
2. Implements the introduced primitives in a Hadoop implementation running on VCL
3. Develops a cost model for estimating the cost of queries expressed in terms of the multidimensional join operator .

Scaling Complex Analytical Processing on Graph Structured Data
Using Map Reduce

by
Radhika Sridhar

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2009

APPROVED BY:

Dr. Tao Xie

Dr. Xiaosong Ma

Dr. Kemafor Anyanwu
Chair of Advisory Committee

DEDICATION

To my parents and sister...

BIOGRAPHY

Sridhar, Radhika was born on February 15, 1983 in Bangalore, India. She obtained her Bachelor's degree in Information Science and Engineering at Dayananda Sagar College of Engineering an affiliate of Vishweshwaraiah Technological University, in May 2005. At the time of writing this, she was working towards her M.S. in Computer Science at the North Carolina State University in Raleigh, North Carolina.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Kemafor Anyanwu for her guidance, helpful suggestions and support in completing this thesis work. I also extend my gratitude to my committee members, Dr. Tao Xie and Dr. Xiasong Ma for their suggestions and valuable comments during the course of my thesis.

I am highly grateful to my family and friends for all their encouragement and support. I would have never come this far without their moral support. I would also like to thank the Semantic Computing Research Group at North Carolina State University for their feedback and suggestions. Special thanks to my friends Padmashree Ravindra and Suchetha M. Reddy for their suggestions and encouragement in completing my thesis work successfully.

TABLE OF CONTENTS

LIST OF TABLES	viii
LIST OF FIGURES	ix
LIST OF ABBREVIATIONS	xi
Chapter 1	1
Introduction	1
1.1 Analytical Processing.....	2
1.2 Data processing on the Semantic Web.....	7
1.3 Research Motivation	17
1.4 Research Contributions	20
1.5 Outline of the thesis	21
Chapter 2	23
Preliminaries	23
2.1 Expressing Complex Analytical Queries using MD-Join	23
2.2 Analytical data processing using parallelism approaches	29
2.2.1 Map Reduce Framework	29
2.2.2 Pig Latin language	33
Chapter 3	39
Implementing MD-join in Map-reduce	39
3.1 Map Function Design.....	40

3.2	Reduce Function Design	42
3.3	MD-Join - Intra-Operator Parallelism	45
	Chapter 4.....	48
	Extending Pig Latin for analytical processing of RDF	48
4.1	Generating Fact Dataset: GFD	50
4.2	Generating Base Dataset: GBD.....	53
4.3	Multi-Dimensional Join: MDJ	56
	Chapter 5.....	67
	Implementation	67
5.1	Execution plan on Hadoop.....	68
	Chapter 6.....	71
	Evaluation.....	71
6.1	Environment.....	71
6.2	BSBM dataset.....	72
6.2.1	Query Execution on BSBM dataset	73
6.2.2	Results	75
6.3	DBLP dataset.....	78
6.3.1	Query Execution on DBLP dataset	79
6.3.2	Results	81
	Chapter 7.....	85
	Related Work.....	85
	Chapter 8.....	89
	Future Work.....	89

Chapter 9	92
Conclusion	92
REFERENCES	93
APPENDIX.....	97
1. Environment setup	98
2. Access to the Hadoop image	102
3. Query execution	106
4. Sample data.....	109

LIST OF TABLES

Table 1 : Relational Representation of the Sales relation	13
Table 2 : RDF representation for the Sales relation.....	14
Table 3: ProdBought.....	16
Table 4: Price.....	16
Table 5: Location	16
Table 6: Fact table representing the Sales data	28
Table 7: Base table created for the table in the fact table	28
Table 8: Base table being updated with the values obtained after performing the aggregation operation	29
Table 9 : Data expressions in Pig Latin.....	37
Table 10: Shows the result obtained after executing GFD.....	53
Table 11 : Subset of the result generated after the GBD operation	55
Table 12 : The results generated after the MDJOIN operation	58
Table 13: Cost analysis for the query execution on BSN dataset.....	75
Table 14: Cost analysis for the query execution on DBLP dataset	81

LIST OF FIGURES

Figure 1: Graphical view of the simple RDF statement.....	9
Figure 2: Graphical representation of the Sales data.....	13
Figure 3 : The architecture of Map-Reduce framework.....	31
Figure 4: Pig Latin architecture	35
Figure 5: Pseudo code for Map Function	41
Figure 6: Pseudo code for Combiner Function	42
Figure 7: Pseudo code for Reducer Function	44
Figure 8: Execution on GFD for Example 4.1	52
Figure 9: Execution of the GBD for the Example 4.1	55
Figure 10: Execution of MDJ for Example 4.1.....	57
Figure 11: Architecture of Pig Latin Language.....	67
Figure 12 : Execution plan of GBD and GFD on Map-Reduce.....	69
Figure 13 : Execution plan for MDJ on Map-Reduce	69
Figure 14: Example subgraph taken from the BSBM dataset	72
Figure 15: Graph shows the cost analysis using the two approaches	78
Figure 16: Example subgraph taken from the DBLP dataset.....	79
Figure 17: Graph shows the cost analysis using the two approaches	83
Figure 18: Screen shot of the Manage Images page	98
Figure 19: Screen shot of the Create an Image page	99
Figure 20: Screen shot of the Connect page	100
Figure 21: Screen shot of the new Reservation page	102
Figure 22: Screen shot showing the Master and the Slave connection details .	103
Figure 23: Screen shot of the GRUNT interpreter.....	106

Figure 24: Screen shot of the sample input data file 109

Figure 25: Screen shot of the sample base data file 110

Figure 26: Screen shot of the output data file 111

LIST OF ABBREVIATIONS

BSBM	: Berlin SPARQL Benchmark
DBLP	: Data Base systems and Logic Programming
GFD	: Generate Fact Dataset
GBD	: Generate Base Dataset
MDJ	: Multi-Dimensional Join
N3	: Notation 3
OLAP	: On-Line Analytical Processing
RDF	: Resource Description Framework
SPARQL	: Simple Protocol And RDF Query Language
SQL	: Structured Query Language
SWETO	: Semantic WEb Technology evaluation Ontology
UDF	: User Defined Function
URI	: Uniform Resource Identifier
WWW	: World Wide Web
W3C	: World Wide Web Consortium
XML	: eXtensible Markup Language

Chapter 1

Introduction

Structured data now constitutes a growing segment of the data being made available on the Web. This trend is due to more organizations appreciating the advantage of making their data available on the Web and also because of the increasingly popular mechanisms for annotating Web content with metadata. These annotation mechanisms range from the informal methods used by applications such as Flickr [12], Delicious[14] Google Co-op[13] that allow users “tag” digital resources with tags of their choice, to the more formal representation schemes such as Microformats, XHTML, Resource Description Framework (RDF)[15], RDF in attributes (RDFa),` Rich/RDF Site Summary (RSS) etc which offer more systematic methods and languages for representing metadata. These more formal mechanisms, particularly RDF – the standard for

metadata exchange on the Web, are gaining broadening adoption because of the promise of potentially enabling reuse, exchange and automatic processing of data. This has created an affinity for RDF in different communities, particularly in scientific research domains where the exchange and sharing of data and the possibility of semi-automatic data integration support is highly desirable. One of the Semantic Web search engines, SWOOGLE[26], now reports that there are several millions of RDF documents currently available on the Web. The implication of this is that, while on the current Web, documents marked up with tags that improve the *presentation* of the document content to enable human understanding, the Semantic Web will have documents in which machines will be able to understand the content on the Web and perform tasks on behalf of users. Further, current generation data processing techniques for the Web will need to be advanced to deal with the structure and semantics in the new Web. In particular, techniques that support more analytical tasks as opposed to the traditional searching and fact-finding will need to be developed for supporting communities such as scientific research communities.

1.1 Analytical Processing

On-Line Analytical Processing (OLAP), in contrast to *On-line transaction processing (OLTP)*, refers to the category of techniques that support efficient

processing of queries demanding aggregations over multidimensional groupings of data. In OLAP, numeric facts about the data called *measures* are represented collectively in a table called the *fact table* and every measure is the value of the attribute associated with the data. Attribute are categorized by a dimension that is derived from the *dimension tables*. The *dimension* provides information about the measure or the attribute. In OLAP, queries aggregate subsets of values in the fact table along multiple dimensions. For example,

Assume that we have a Customer relation (CustID, CustName) (typically called a *dimension table*), a Sales relation (CustID, ProdID, Price, Location) (typically referred to as a *fact table*) that relates customers to products that they *bought* and the *price* paid and *location* in which the sale occurred. We may want to compute total *sales* amounts when grouped by all combinations of *product*, *month* and *state*. This results is a query with aggregations (total sales) over eight different groupings for every combination of *product*, *month* and *state* (i.e. *none*, (*product*), (*month*), (*state*), (*product & month*), (*month & state*), (*state & product*) and (*product & month & state*). Such queries are fundamental to analytical tasks in business and financial applications. However, investigative applications such as in scientific research domains, often require more ad-hoc analytical queries as well as scientific research domains but are challenging and cumbersome to express and evaluate efficiently. Some special operators

such as the CUBE, ROLLUP, etc were added to SQL which makes reporting-style queries. However, ad-hoc analytical queries tend to be more complex and are not easily expressible they require multiple aggregations over different groups. For example, suppose we want to gain some insight into the buying patterns in a particular region, we might want to compute for every *customer*, the total *amount* of their purchases in either of the *states*, say “NY” or “NJ”. This is called a “*pivoting*” query whose result is a relation (CustID, Total_NY, Total_NJ). Since this requires computation of *sales* in those states for every *customer*. Expressing such queries using traditional relational database approach would require two subqueries (one for each location) to compute the total amount of sales for the location, then two outer joins to Customer table to assemble the final result. Such a query expression is cumbersome and optimizers don’t often select the best execution plan for them. Consider the Sales schema shown below [7]:

Sales(CustID, ProdID, Price, Location, Month, Year), to compute the average for each customer who purchased the products in “NY”, “NC” and “NJ”.

Evaluating such a query using the regular relational database operator would primarily require executing the three subqueries, each query to compute the per

customer sales in NY, NC and NJ respectively. The result gives a list of all the customers, whether or not they made any purchases in these states. We need another subquery to select all the unique customers. Finally, we need four outer joins to attach the sales to the customers in NY, NC and NJ locations.

A key observation made in [7] is that, there exists a tight coupling between the grouping operations and the aggregation function that needs multi-pass aggregation.

1.2 Challenges of Analytical Processing on RDF

Scalability - The issue of efficient processing of data at Web scale is still a primary concern for search engine companies as datasets range to terabytes of data. Parallel processing seems to be one promising approach for processing data at a Web scale. Traditional approaches that use high-end parallel database systems with highly specialized architectures such as Teradata, Tandem, NCR, Oracle-n CUBE, and RAC or OLAP servers such as Microsoft OLAP servers, SAS OLAP server are not cost effective and easily adoptable strategy. These high end systems, though quite capable of handling data stores at enterprise scale, are not designed for the Web scale processing and are too expensive to be a practical alternative for supporting the Web scale processing. Alternatively, there are leading efforts to develop platforms that enable parallel processing of Web data. The winning and popular approach, pioneered by

Google, is the *Map-Reduce* [18] framework that has its roots in functional programming languages. Further, Apache's release of an open source version of Map-Reduce called *Hadoop* [1] derive from the Map-Reduce approach. These platforms are designed to run parallel programs on a computational cluster of commodity grade machines, a paradigm popularly known as *cluster computing*. Further, a language *Pig Latin* [6] is built on top of Hadoop which is an open source implementation of the Map-Reduce Framework. Pig Latin is an algebraic dataflow language that expands the scope of primitives to enable the reuse of common code fragments and provides the opportunity for applying query optimization techniques. However, these approaches currently focus on supporting the simple data processing tasks with the limited support for semi structured or graph structured data such as RDF.

In order to execute ad-hoc queries on RDF datasets, before performing any aggregation operations, we need to reassemble all the tuples with the related predicates. A series of join operations are required to reassemble the tuples. After reassembling the tuples, multi-pass aggregations need to be computed which requires repeated processing on the same set of tuples with slightly different computations, thus making the execution of these queries inefficient. Our aim is to provide an efficient approach that can perform analytical querying efficiently on semi structured datasets like RDF.

Various operators like the MD-Join, GMD-Join are proposed in the relation database to perform efficient complex analytical query executions.

There are various areas that require performing analytical queries on the Semantic Web. Fields like biomedical research, bioinformatics, etc., aim in turning Semantic Web into practical applications that involve performing complex analytical queries on the data to enhance the research ideas leading to new innovations and discoveries

With such rapid growth rate of the semantic data, there is an increasing need for a scalable approach to process these data. There are various scalable parallel processing approaches like the Map-Reduce framework, Pig Latin language that executes over a Map-Reduce framework and so on. But these approaches currently process simple data efficiently. Executing complex analytical queries on structured semantic data using these frameworks are yet to be researched.

1.3 Data processing on the Semantic Web

A fundamental data model for the Semantic Web is called the *Resource Description Framework (RDF)* [15]. In RDF, a simple statement is a triple of the form $\langle \textit{Subject}, \textit{Predicate}, \textit{Object} \rangle$, where *Subject* can be any resource that needs to be described. *Predicate* indicates the property associated with that

subject and the *Object* holds the value for that property. The triple representation of RDF can be used to describe any concept, relationship or an object that exists in the universe. In RDF the resources are identified using simple Web identifiers called the Uniform Resource Identifiers (URI). This enables to represent resources and their properties as graphs of nodes and arcs representing their properties. For example, consider a general statement:

“Customer Joe purchased a Dell laptop”. The RDF representation of the statement is as shown below:

```
< http://examples.com/Cusomer#Joe,  
http://examples.com/purchased,  
http://www.dell.com/produce#Dell_laptop >
```

In the above example, “*http://examples.com/Cusomer#Joe*” represents the subject, “*http://examples.com/purchased*” represents the property and the object is represented by “*http://www.dell.com/produce#Dell_laptop*”. The RDF statements can also be represented as a labeled graph connecting resources where the labeled edges represent the properties between the resources. Figure 1 shows the graphical representation of the RDF statement.

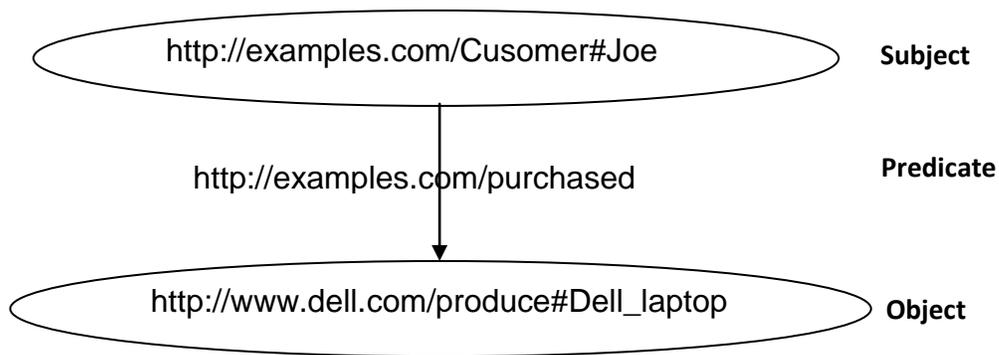


Figure 1: Graphical view of the simple RDF statement

Further, RDF is a conceptual model with different serialization formats. Some concrete formats of representation are: *XML RDF*, *Notation 3*, *N-Triples*, and so on. In this section we briefly describe the Notation 3 syntax that is one of the simplest and widely used formats of RDF representation [26]. In this notation, the subject, predicate and the object are URI's enclosed with in "<" and ">" symbols. The end of each line or a triple is denoted by ".". The syntactical form of Notation 3 is as shown below, where subject, predicate and object are *atoms*. An atom can either be an URI, an URI abbreviation, a blank node or a literal.

<subject><predicate><object> .

For example,

- 1) <http://example.org/#Joe> <http://example.org/#Type>
<http://example.org/#Customer> .
- 2) <http://example.org/#PO12> <http://example.org/#loc>
<http://example.org/#NC> .
- 3) <http://example.org/#PO12> <http://example.org/#price> <35> .

Representing the data using such RDF syntax provides some structure for the contents on the Web that makes the Web appear as a globally linked database of triples as opposed to just a network of unstructured documents. This provides an environment for the Web contents to be queried and analyzed to a degree similar to what has been achieved with structured data. For example, Search engine companies are actively investigating on techniques for analyzing the massive amount of search log, click stream and web graph data that they collect.

The World Wide Web Consortium (W3C) recommends *SPARQL Protocol and RDF Query Language (SPARQL)* [16] for querying RDF data. The SPARQL language supports querying RDF graphs and is designed to execute queries using a combination of *triple patterns*, *variables* and *constants*. Variables in

SPARQL are represented using symbol “?” or “\$” that prefix the variable name.

For example, consider a RDF statement as shown below:

```
<http://example.org/#PO12> <http://example.org/#price> <35> .
```

A query to find the price of Product PO12 can be written in SPARQL as:

```
SELECT ?price
WHERE {
  <http://example.org/#PO12>
  <http://example.org/#price> ?price . }
```

The result of the above query returns value 35. Simple queries can be executed using the SPARQL query language. More complex queries can be formed by combining multiple triple patterns to form graph patterns using combination operators. Currently, it is not possible to express queries that require grouping and aggregations operations using SPARQL because the language does not support these operations. . Some systems like OpenLink Virtuoso [30], ARQ [4] , etc., extend SPARQL with SQL like aggregate and grouping functions. However, even with such systems in these systems, executing complex queries does not result in efficient results. Since, queries with multiple groupings and aggregations require each aggregation function with the corresponding

grouping attribute to be executed as a separate subquery. Hence each subquery requires scanning the table at least once. Performing multiple scans result in expensive computations.

Further, the structure of the RDF data creates some challenges in performing complex queries on the Web content. The challenges being an n-ary tuple in a relational scheme contains all related data values in a single unit. Thus each tuple is independent of the other within the input file. Grouping and aggregation operations are executed on these tuples which means that such operations are performed at the level of related data values. However, when we consider the RDF data model, each tuple is a combination of the subject, predicate and the object. Thus an n-ary relational tuple would be spread across a set of (n-1) RDF triples. For example, Figure 2 shows the graph representation for two tuples of the Sales relation

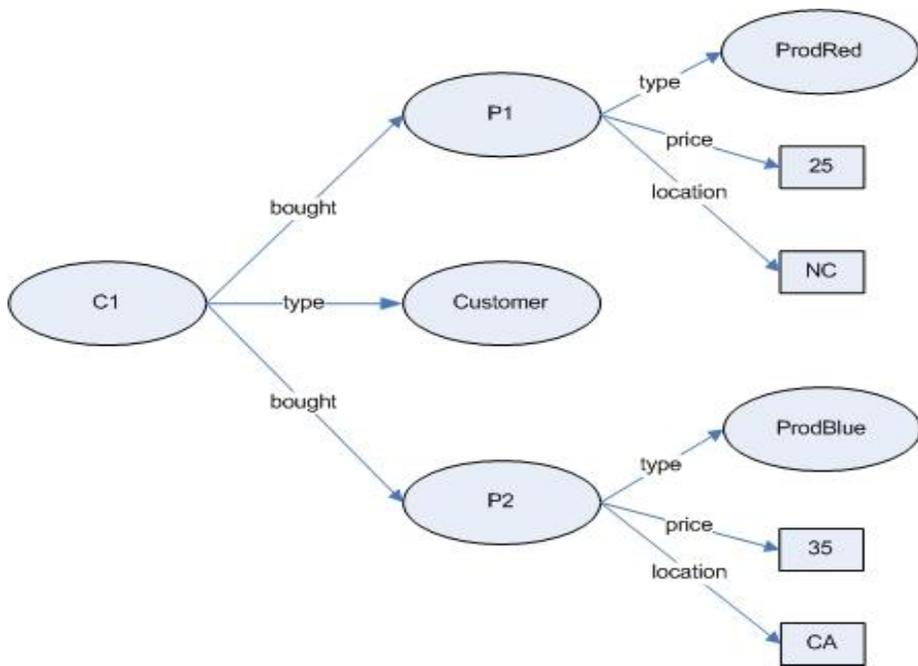


Figure 2: Graphical representation of the Sales data

Representing this graph in a relational database will result in a table having attributes *Customer*, *ProdBought*, *Price* and *Location*. The data in Table 1 corresponds to the data in the graph.

Table 1 : Relational Representation of the Sales relation

Customer	ProdBought	Price	Location
C1	P1	25	NC
C1	P2	35	CA

The equivalent RDF representation for the same data is shown in Table 2. Comparing Table 1 and 2, we see, a single tuple with four attributes in the relational representations is shredded into three tuples of subject, predicate and object in the RDF representation.

Table 2 : RDF representation for the Sales relation

Subject	Predicate	Object
C1	Bought	P1
P1	Price	25
P1	Location	NC
C1	Bought	P2
P2	Price	35
P2	Location	CA

Due to the shredding of the tuple in the RDF representation, operations like grouping or aggregation will require intensive self-joins over the same set of triples, resulting in additional cost during query executions. For example, consider a simple query –

To find all customers who bought products from location “NC”.

Expressing the above example in relational algebra:

$$\pi_{\text{Customer}} (\sigma_{\text{Location} = \text{"NC"}} (\text{Sales}))$$

In order to obtain the same result from the RDF dataset, we need to express the above example in relational algebra as:

$$\pi_{\text{subject}} (R \bowtie_{R.\text{subject} = S.\text{object}} S)$$

Where,

$$R = \sigma_{\text{predicate} = \text{"Location and object} = \text{"NC"}} (\text{Sales})$$

$$S = \sigma_{\text{predicate} = \text{"bought"}} (\text{Sales})$$

Further, a popular approach for efficient management of the RDF data is commonly known as the *vertically partitioned approach*, in which the dataset containing the RDF data is partitioned into “n” unique datasets based on the distinct tuple properties. Every dataset contains all the tuples corresponding to one unique property. In this approach, the tuples are still shredded, but are stored in separate files based on their properties. Thus performing any operations on these datasets still requires reassembling of the data, but in this case reassembling of the data can be performed by executing join operations

on different datasets, rather than the self join operation performed as in the earlier approach. For example,

Consider the above examples, suppose the Sales data shown in Table 2 is partitioned using the vertically partitioning approach, then the Sales data would be represented as follows:

Table 3: ProdBought

Subject	Object
C1	P1
C1	P2

Table 4: Price

Subject	Object
P1	25
P2	35

Table 5: Location

Subject	Object
P1	NC
P2	CA

Table 3 represents the data that has predicate value *ProdBought*. Similarly Table 4 contains all the data that has predicate value *Price* and Table 5 contains data with predicate value *Location*. In order to obtain the same result on the vertically partitioned RDF dataset the query shown in the above example, we can represent the query in relational algebra as follows:

$$\pi_{\text{subject}} (R \bowtie_{R.\text{object} = S.\text{subject}} S)$$

Where,

$$R = \sigma_{\text{object} = \text{"NC"}}(\text{Location})$$

$$S = \sigma_{\text{predicate} = \text{"bought"}} (\text{Sales})$$

The above example shows the need to reassemble the related RDF tuples before executing any queries on them. Table 2 also shows how the RDF format, combines data and metadata within their representation. The attributes *ProdBought*, *Price* and *Location* in the Table 1 is the actual data in the RDF dataset. This adds additional complexity in querying RDF data, since it is necessary to check for the correct predicate before computing any aggregation or grouping operation. These challenges show the need for an approach to query the RDF data that is similar to the relational database queries considering the structure of the RDF datasets.

1.4 Research Motivation

Enable analytical querying on RDF datasets:

Ad-hoc analytical queries tend to be more complex as they require multiple aggregations over different groups or viewing the results from different dimensions. For example [7],

Consider the Sales schema shown below:

Sales(CustID, ProdID, Price, Location, Month, Year), Suppose, to compute the average sales value of each customer who purchased the products in "NY", "NC" and "NJ".

Evaluating such a query using the regular relational database operator would primarily require executing the three subqueries, each query to compute the per customer sales in NY, NC and NJ respectively. The result gives a list of all the customers, whether or not they made any purchases in these states. We need another subquery to select all the unique customers. Finally, we need four outer joins to attach the sales to the customers in NY, NC and NJ locations. This example shows the need for multi-pass aggregation. In order to execute ad-hoc queries on RDF datasets, before performing any aggregation operations, we need to reassemble all the tuples with the related predicates. A series of join operations are required to reassemble the tuples. After reassembling the tuples, multi-pass aggregations need to be computed which requires repeated processing on the same set of tuples with slightly different computations, thus making the execution of these queries inefficient. Our aim is to provide an efficient approach that can perform analytical querying efficiently on semi structured datasets like RDF.

A scalable approach for RDF data processing: The issue of efficient processing of data at Web scale is still a primary concern for search engine companies as datasets range to terabytes of data. Parallel processing seems to be one promising approach for processing data at a Web scale. Traditional approaches that use high-end parallel database systems with highly specialized architectures such as Teradata, Tandem, NCR, Oracle-n CUBE, and RAC or OLAP servers such as Microsoft OLAP servers, SAS OLAP server are not cost effective and easily adoptable strategy. These high end systems, though quite capable of handling data stores at enterprise scale, are not designed for the Web scale processing and are too expensive to be a practical alternative for supporting the Web scale processing. Alternatively, there are leading efforts to develop platforms that enable parallel processing of Web data. The winning and popular approach, pioneered by Google, is the *Map-Reduce* [18] framework that has its roots in functional programming languages. Further, Apache's release of an open source version of Map-Reduce called *Hadoop* [1] derive from the Map-Reduce approach. These platforms are designed to run parallel programs on a computational cluster of commodity grade machines, a paradigm popularly known as *cluster computing*. Further, a language *Pig Latin* [6] is built on top of Hadoop which is an open source implementation of the Map-Reduce Framework. Pig Latin is an algebraic dataflow language that expands the scope

of primitives to enable the reuse of common code fragments and provides the opportunity for applying query optimization techniques. However, these approaches currently focus on supporting the simple data processing tasks with the limited support for semi structured or graph structured data such as RDF.

1.5 Research Contributions

In the earlier section, we have discussed the issues and the challenges involved in analytical querying on RDF datasets. Based on these challenges, we aim to contribute the following-

- Clearly introduce the problem of analytical data processing on RDF datasets
- Propose an approach for achieving the scalable processing of the non-trivial analytical tasks on RDF datasets that is based on an efficient multidimensional query operator called the *MD-Join* and parallel query processing on an extended Map-Reduce framework.
- Propose an approach for implementing the multi-dimensional join in a Map-Reduce framework
- Propose an extension to the Pig Latin dataflow language that includes the structural and semantic query expressions that are necessary for querying RDF data, provide query primitives for

reassembling related RDF data values and define the inputs necessary for MD-join operator. Further, we show how this extended Pig Latin language compiles into the Map-Reduce workflows with the enhanced MD-joins.

1.6 Outline of the thesis

- Chapter two discusses the challenges involved in expressing the complex analytical queries and introduces the existing multi-dimensional join operator. Further in this chapter, various parallelism approaches for data processing are discussed.
- Chapter three discusses in detail the implementation of the multi-dimensional join operator on a Map-Reduce execution framework.
- Chapter four explains how Pig Latin language can be extended to provide new operators that can perform analytical processing on RDF
- Chapter five discusses the execution plan for these extended operators in terms of the Map-Reduce functions
- Chapter six shows the results of a few query executions using the extended operators

- Chapter seven discusses the related work and chapter eight highlights the possible future work.
- Chapter eight finally gives the conclusion.

Chapter 2

Preliminaries

2.1 Expressing Complex Analytical Queries using MD-Join

In the examples seen in section 1.1, we have observed that the complex ad-hoc queries involve multiple aggregations over different sets of grouping values. In the relational database, to perform a set of aggregation operations on different grouped attributes, every aggregation operation has to be performed on one set of grouping attributes independently and then these results have to be combined. This tight coupling between the aggregation function and the grouping attributes results in a series of join and union operations. The multiple joins and unions restrict the optimization of these queries resulting in inefficient query executions. For example [6],

Example 2.1 - we would like to compute the total number of products having sales between the average sale of the previous month and the average sale of the next month, for all combinations of the product and the month for the year “2000”.

Evaluating such a query using traditional database operators would mean to filter out all records for which the condition $\text{year} = 2000$ is not valid. For tuples where the filter condition is true, a GROUPBY operation is performed over all the *product* and *month* combinations. We would then need to compute aggregates from tuples that are outside of each group (the previous and next month’s average sale). Using these results, the final aggregation value can be computed. This example shows how cumbersome it is to express such complex queries using the ordinary SQL operators. New operators like CUBE, PIVOT, etc cannot be used in these queries as the computation is more complex than a simple aggregation over multi-dimensions. These observations were made in [8] and an operator, the *MD-Join* operator, that allows the queries to decouple the grouping and the aggregation functions, was proposed. In this operator, a *Base table is constructed*, that is a container table holding all the combinations of the key sets for which the aggregation has to be computed.

The actual data in the relational table represents the fact table. The following is a formal definition for the MD-Join operator:

Definition: Let B and R be relations, Θ is a set of conditions involving the attributes of B and R, I is a list of aggregation functions that needs to be computed, $I = (f_1, f_2, f_3, \dots, f_n)$ over attributes $c_1, c_2, c_3, \dots, c_n$ of R. We define a new relational operator between B and R, called *MD-Join*, defined as:

$MD(B, R, I, \Theta)$ is a relation with schema B, $f_1_R_c_1, f_2_R_c_2, \dots, f_n_R_c_n$, whose instance is determined as follows. Each tuple $b \in B$ contributes to an output tuple B, such that:

- Table B is augmented with as many columns as the number of aggregate functions in I. Each column is named as $f_i_R_c_i, i = 1, \dots, n$
- For each row r of table B we find the set S of tuples in R that satisfy Θ with respect to r, i.e. when B's attributes in Θ are replaced by the corresponding r's values. Then, the value of

column $f_{i_R_c_i}$ of row r is the $f_i(c_i)$ computed over tuples of S , $i = 1, \dots, n$.

B is the *base table* created and R is the *fact or detail table* that holds the collection of related tuple values e.g. the Sales relation. The semantics of the MD-join operator is designed in such a way that the sequence of MD-joins can be combined together, thus making the execution of complex ad-hoc query cost efficient.

Expressing the above example using the MD-join operator we get:

$MD(MD(MD(B, Sales, AVG(sale), \Theta_1), Sales, AVG(sale), \Theta_2), Sales, AVG(sale), \Theta_3)$

Where $\Theta_1 : Sales.cust = cust$ and $Sales.state = "NY"$,

$\Theta_2 : Sales.cust = cust$ and $Sales.state = "NJ"$,

$\Theta_3 : Sales.cust = cust$ and $Sales.state = "NC"$,

And B is the table generated using a simple query of the kind, "*select distinct cust from sales*".

The above example shows how the analytical query can be executing without performing additional joins to combine the results of the aggregation operations.

The MD-join operator separates the tight coupling that exists between grouping and aggregation attributes and hence makes the query execution efficient. Due to this separation, it is possible to compute the aggregation value for all combination of the attributes at once instead of performing the aggregations for each combination of attributes separately which will require additional scanning of the table.

Algorithm:

```
Scan R, and for all tuples t in R{  
    For all rows r of B, check if condition  
     $\Theta$  is satisfied with respect to r and t.  
    If yes, update r's aggregate columns  
    appropriately.  
}
```

The above algorithm shows the computation of the MD-join operator. This operator captures the semantics of the user's need, more accurately as shown in the above example. To understand the execution of the MD-join algorithm consider the following tables. Table 6 is the fact table consisting of the Sales data. Table 7 is the base table generated from the data present in the fact table.

Table 8 shows the result generated after the execution of the MD-Join operator on the fact table. When a match is found between the fact and the base table the corresponding aggregation is computed and the value is updated in the base table as shown in Table 5.

Table 6: Fact table representing the Sales data

Cust	Prod	Month	Year	Price
1290	PRD1937	Feb	2008	1982.89
1291	PRD9436	Dec	2007	899.98

Table 7: Base table created for the table in the fact table

Prod	Month
PRD1937	Feb
PRD1937	Dec
PRD9436	Dec

Table 8: Base table being updated with the values obtained after performing the aggregation operation

Prod	Month	Sum(Price)
PRD1937	Feb	1982.89
PRD1937	Dec	89.90
PRD9436	Feb	32.2 100.18
PRD9436	Dec	NULL
PRD9490	Jan	NULL

2.2 Analytical data processing using parallelism approaches

2.2.1 Map Reduce Framework

Map-Reduce is a programming model designed to perform distributed computation on clusters of computers. This framework is designed on the idea of the functional programming technique, where the computation of the tasks is performed by various functions. This framework defines two basic primitives called the *Map* function which groups together the related data and the *Reduce*

function which performs any kind of computation or aggregation operation on the groups. These Map and the Reduce functions are simple function prototypes which needs to be implemented by the users as per the user requirements. To use this paradigm for data processing, tasks need to be mapped into the Map and the Reduce functions. Data processing using this framework is particularly suited for tasks that can be casted as group-by-aggregation. The execution of the tasks in the map and reduce functions are independent of each other as each of these functions perform one specific task. Hence these functions can be executed on different processors at different times. This allows the processing of the tasks in these functions to be parallelized. Figure 3 shows the architecture of the Map-Reduce framework. The framework consists of one master process and multiple task processors running in parallel to perform either a Map function or a Reduce function. The master process accepts the input data and splits the data into smaller chunks and assigns each chunk to one of the available processors. The map function reads each line in the data, generates a key and returns the result as a *<key, value>* pair. After all the map processors have completed their tasks, the master function assigns the tasks for the processors to perform the reduce function. This function sorts and merges all the intermediate keys generated by the map function and computes the results. Any sort of aggregation that needs to be

computed can be performed in the reduce function. The keys and the type of values generated for the map and the reduce functions are of the following types:

Map (k1, v1) \Rightarrow list (k2, v2)
Reduce (k2, list (v2)) \Rightarrow list (v2)

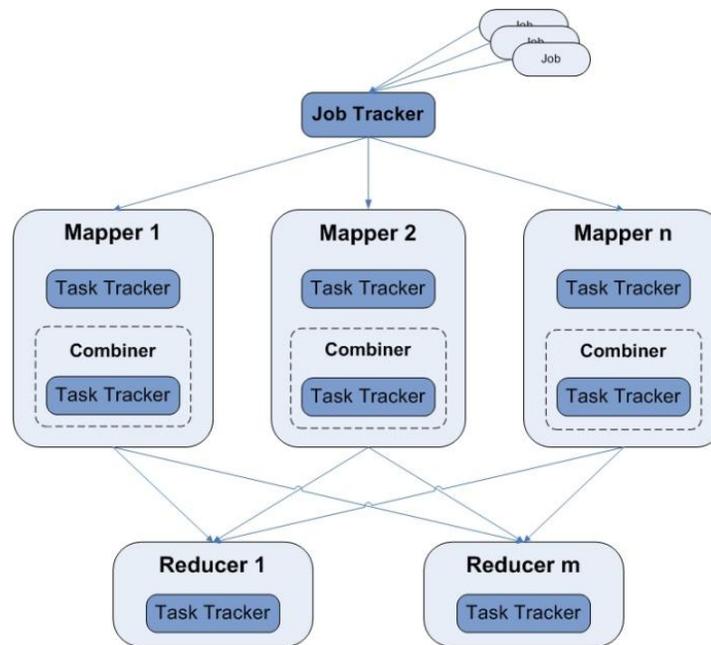


Figure 3 : The architecture of Map-Reduce framework

Consider a simple example from [18] that counts the number of occurrences of each word in a large document using the Map-Reduce framework. The master process reads the input file, breaks it into chunks of smaller size and assigns the chunks to the available processors to execute the Map function. Each processor performing the Map function receives the <key, value> pair where the *file name* is the key and the file data is the value in this case.

In each map process, the data is read and the output is a list of <key, value> pairs, where the key is all the words present in the document and the value is assigned as “one”. The processors performing the reducer function, will receive <key, a list of value> pairs. We can now add up all the values corresponding to a key and return the number of times that key (i.e. the word) occurs. Hence the output of the reducer will be the list of all the keys and the number of occurrences of the word in the document. This example shows how we can parallelize the simple task of counting the number of occurrences of each word in a file. However, more complex processing can be achieved by customizing the map and the reduce functions.

Further, the Map-Reduce framework is designed to work with a single dataset. However, in many situations it is required to perform join operations over two or more data sets. The method for bypassing this limitation of a single dataset includes the *Map-Side Join* or a *Reduce-Side Join*. In the Map-Side Join

approach, we first perform a map function, skip the reduce function but generate the keys such that in the next cycle of map all the related data have the same key. This approach requires one additional cycle of Map-Reduce data flow for each of the JOIN operations performed. The other alternative is to generate a flag representing each dataset. Based on this flag we can perform the JOIN in the reduce function. This is known as the Reduce-Side Join. These two approaches to perform JOIN operations are not very efficient because they require an additional map and reduce phase in order to perform the JOIN operation. Hence an extension of the Map-Reduce framework is the *Map-Reduce-Merge* [12] framework. In this programming model, an additional merge phase is added, where the JOIN operation is performed between the two sets of the data that are obtained from the two different Map-Reduce cycles.

2.2.2 Pig Latin language

Map-reduce programming model has proven to be an efficient approach for the data processing task, but with a few limitations such as:

- Rigid data flow between the Map and the Reduce functions. In the above section, we observed, when implement the Map-Side join we execute one cycle of the Map and Reduce function with no operation in

the Reduce function. We cannot eliminate this Reduce function due to the rigid data flow between the two functions. Hence we execute the Reduce function after the execution of the Map function but perform no operation in the Reduce function.

- Framework's primary reliance on the customized functions that provide limited opportunity for an automatic optimization and reuse of the code.

These limitations are the motivation for the development of the new dataflow language known as the *Pig Latin*. The main goal of the Pig Latin language was to achieve a sweet spot between the declarative style of the languages like SQL and the low level procedural style of the Map-Reduce programming. In order to achieve this, Pig Latin provides a set of predefined functions and query expressions that can be used to describe the data processing tasks. Along with these pre-defined functions, the language also allows the user to define their own functions called the *User Defined Functions* (UDF). Figure 4 shows the architecture of the Pig Latin language.

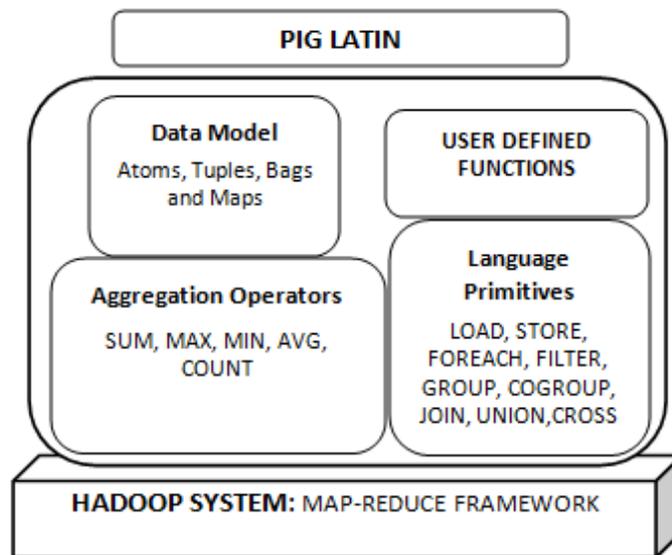


Figure 4: Pig Latin architecture

The data model for a Pig Latin consists of an *atom* that holds a single atomic value, a *tuple* that holds a series of related values, a *bag* that forms a collection of the tuples and a *map* that contains a collection of the key value pairs. A tuple can be nested into an arbitrary depth. The basic primitive functions of the Pig Latin language are: the LOAD function that can determine what the input file is and how the file has to be read. The FOREACH function is used as an iterator to loop through the collection. The FILTER primitive discards all the tuples for which the condition does not hold. The GROUP operator collects the similar data

records within a given data set. The COGROUP operator is similar to the GROUP operator but focuses on grouping together the similar data from different data sets. The JOIN function is used to merge the data from two different datasets. Other common commands similar to the SQL commands are the UNION, DISTINCT, ORDER, CROSS, AVG, SUM, MIN, MAX and so on. STORE is used to get the results stored in an output file. In addition to these primitive, Pig Latin provides a library of UDF's – *User Defined Functions*. The limitation of UDF is that the users will be responsible for the efficiency of their programs and they have to specify how the functions have to be parallelized.

For example, consider the Sales schema discussed in chapter 1,

Compute the average sales within each location, where the number of purchases in that location is greater than 3. A Pig Latin program for this scenario, using the above mentioned operators is as follows:

```
groups = GROUP Sales BY Location;
```

```
group_count = FILTER groups BY COUNT(*) > 3;
```

```
output = FOREACH group_count GENERATE AVG(price);
```

The above example shows the sequence of steps in a Pig Latin program, which is much like any programming language. Each line in the program performs a single data transformation. These transformations in every step are fairly high level, resembling the SQL, e.g. FILTER, GROUP, etc. Along with these SQL like operators, Pig Latin provides a wide variety of data expressions and different kinds of nested tuples for the data storage. This language provides a flexible approach for accessing the data from these nested tuples. For example, consider a tuple t with fields' f_1 , f_2 and f_3 , where t is defined as shown below:

$$t = \left('Cust1', \left\{ \begin{array}{l} ('NC', 25) \\ ('CA', 35) \end{array} \right\}, 'Prod123' \right)$$

Table 9 : Data expressions in Pig Latin

Expression Type	Example	Value for t
Field by position	$\$2$	'Prod123'
Field by name	f_1	Cust1
Projection	$f_2.\$1$	$\left\{ \begin{array}{l} (25) \\ (35) \end{array} \right\}$
Function Evaluation	$SUM(f_2.\$1)$	$25 + 35 = 60$

Table 9 shows some examples of the expression type in Pig Latin and also how these expressions operate. The flexibility provided in these expressions allows the user to perform various kinds of operations on the data.

Chapter 3

Implementing MD-join in Map-reduce

In section 2, we briefly defined the MD-JOIN operator and its advantages over the other OLAP operators like CUBE, GROUPBY and so on. In this section, we describe how the MD-JOIN algorithm can be implemented to execute as a Map-Reduce function using the Hadoop system. The Hadoop system consists of a *Job Tracker* which acts as a master process, reads the input data, divides the input dataset into chunks of equal size and assigns them to each of the *Task Trackers*. *Task Trackers* are the processors that are designed to perform the Map or the Reduce functions. In this implementation one cycle of the Map-Reduce is executed to generate the base dataset from the dataset given by the user. The master process divides the task of processing the fact dataset and the base dataset into subtasks and assigns each of the subtasks to one of the

available map process. Each subtask that is processed by one of the Map functions generates a list of intermediate results. After all the input data is processed and the intermediate results are generated, the master process assigns the results to the available processors to perform the reduce jobs. The results of the reduce jobs are written to an output file, which is the result of the MD-join operation. The following sub sections show the Map and the Reduce functions for MD-join implementation.

3.1 Map Function Design

The Job Tracker assigns each of the Map functions with a chunk of the dataset, where each tuple in the dataset is of the form <Subject, Property, Object>. The pseudo code below shows the implementation of the Map function. The Map function reads every tuple and generates the <key, value> pairs as an output, where the value is a map of the predicate and the object corresponding to the subject. Hence the Map function returns a list of <Subject, <Predicate, Object> > pairs.

Figure 5 shows the pseudo code for the Map function.

```
Map ( String Map, String Value)
//Key : File Chunk Name
// Value : chunk of data
For each line in the value
// line is of the form <Subject, Property, Object>
Output(<Subject, <Property, Object>>
```

Figure 5: Pseudo code for Map Function

The Combine function is a sub-routine that is implemented within the Map function and combines the related results of the Map. This function is executed by the framework after the executing of the Map functions. All the values having the same key are grouped together into a collection. If the user defined query has any filter conditions defined on the dataset, then the corresponding <key, Collection of values> are filtered out in this function. The pseudo code for the combine function is shown below.

```
Combiner (Collection Output)
//Output has the list of <subject, <Property, Object>>
records
FOREACH Subject in the Output
Get all the records with the same Subject
IF Filter Condition
IF Aggregation is on a Multidimensional Key
Key = generate composite key based on the
attribute value
Value = <Property of the Attribute, Value of that
Property>
Else
Key = Subject
Value = <Property, Value>
End IF
End IF
Output (<Key, [Value]>)
```

Figure 6: Pseudo code for Combiner Function

In [9] the authors show that the tuples for which the filter condition is not true will never be considered by the MD-Join and hence these tuples can be eliminated from the dataset. Thus MD (B,R,I, Θ), where Θ involves the attributes of R is equivalent to MD(B, Selection on Θ (R), I, Θ); By eliminating the data records for which the filter condition is not true, we are reducing the number of records to be processed in the reduce function, thus increasing the efficiency.

3.2 Reduce Function Design

Set of keys and the collection of values are the input to the reduce function. Each reducer will have a set of the fact tuples and the corresponding base

tuples. Base tuples are of the form $\langle \text{key}, \langle \text{BASE}, \text{NULL} \rangle \rangle$, where “BASE” is used as a flag to perform the join operation. The fact tuples are of the form $\langle \text{key}, [\langle \text{property}, \text{value} \rangle]$. For examples,

Example 3.2.1: To compute the number of each product purchased.

The base tuples for this example will be of the type:

$\langle \text{PROD123}, \langle \text{BASE}, \text{NULL} \rangle \rangle$

$\langle \text{PROD342}, \langle \text{BASE}, \text{NULL} \rangle \rangle$

$\langle \text{PROD566}, \langle \text{BASE}, \text{NULL} \rangle \rangle \dots$

Consider the following to be the set of fact tuples containing the product id and the product purchase information.

$\langle \text{PROD123}, \langle \text{LOC}, \text{NC} \rangle \rangle$

$\langle \text{PROD342}, \langle \text{LOC}, \text{NY} \rangle \rangle$

$\langle \text{PROD566}, \langle \text{LOC}, \text{NC} \rangle \rangle$

$\langle \text{PROD934}, \langle \text{LOC}, \text{NC} \rangle \rangle \dots$

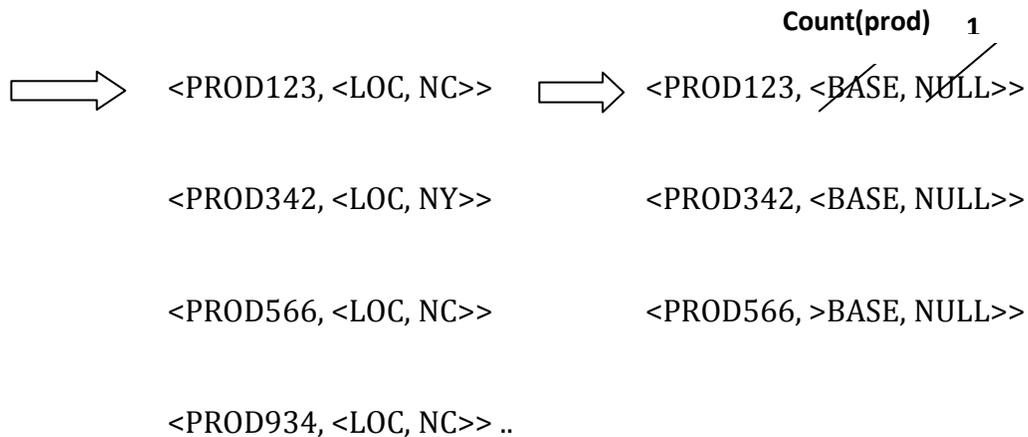
All the properties and the corresponding values for the keys are collected together. The algorithm for computing the aggregation is shown below. When a match is found between the base and the fact tuple, the aggregation operation is computed and the base tuple is updated with the value computed. Figure 7 shows the pseudo code for the reducer function.

```

Reduce(String Key, Iterator Value)
    // The MD-Join Algorithm is implemented in this function
    FOREACH occurrence of the user defined condition in
    the fact set
        IF Fact.key == Base.key
            compute the Aggregation Function
        And update the Base dataset, by replacing the
        NULL value in it.
    
```

Figure 7: Pseudo code for Reducer Function

Executing the Reduce function on the data shown in Example 2.3.1, we obtain the following:



The example above shows the execution of the reduce function on the dataset, and how the aggregation values are updated on the Base dataset.

As discussed in section 2.1, MD-join is designed to separate the tight coupling between the grouping attributes and the aggregation functions. Due to this decoupling it is possible to perform the grouping operation and the aggregation in different functions. Since the grouping and the aggregation operations are independent of each other, we can perform the grouping and the aggregation operations in the Map and the Reduce functions respectively. Further, in section 3.3, we show how the MD-join operator can be executed in parallel.

3.3 MD-Join - Intra-Operator Parallelism

In section 2.1 we discussed MD-Join for the sequential execution of the data. In this section, we will see how the MD-Join operator is amenable to parallelism by leveraging the results given in [9] which state the following:

Observation 3.3.1: If B and R are relations, B_1, B_2, \dots, B_m a partition of B , I is a list of aggregate functions over columns of R and Θ is a set of conditions involving attributes of B and R , then:

$$MD(B,R, l, \Theta) = MD(B_1,R, l, \Theta) \cup MD(B_2,R, l, \Theta) \cup \dots \cup MD(B_m,R, l, \Theta)$$

This observation states that the query using MD-JOIN can be parallelized by dividing the base dataset across the processors and executing the MD-Join algorithm on each of them in parallel. This reduces the execution time for processing the complex queries, but each processor still needs to have the entire fact data set and iterate through it completely to check if there is a matching key found for computing aggregation.

Based on the above conclusion the following observations were made,

Observation 3.3.2: In $MD(B,R,l,\Theta)$, B can be partitioned into $B_1 \cup B_2 \dots \cup B_n$ where $B_i = \sigma_i(B)$, where σ_i is a range selection based on the attributes on B. Similarly R can be partitioned into $R_1 \cup R_2 \dots \cup R_n$ where $R_i = \sigma_i(R)$, where σ_i is a range selection based on the attributes on R. The same selection function is used for both base and the fact table partitioning in such a way that the same range of selection is performed. Hence,

$$MD(\sigma_i(B), R, l, \Theta) = MD(\sigma_i(B), \sigma_i(R), l, \Theta)$$

This observation states that the query using MD-JOIN can be parallelized by dividing both the base dataset and the fact dataset across the processors such that every processor gets the same range of base and fact data. Thus the MD-Join algorithm can be executed on each of them in parallel for the subset of the fact and the base data.

This section shows how the MD-join operator can be implemented as a map-reduce function. Since this is a low level implementation of the operator, any customized computation that needs to be done, requires the user to change the Map and the Reduce functions. As discussed in section 2.2, the user customized code does not allow the efficient optimization. In the next section we provide a set of new operators for Pig Latin language to perform complex analytical querying.

Chapter 4

Extending Pig Latin for analytical processing of RDF

Pig Latin provides various operators like the JOIN, FILTER, GROUP and COGROUP which can be used to support the basic analytical queries.

Example 4.1: Consider the data shown in Table 2 to get all the list of all the customers who bought products in location NC, we need to execute the following queries in Pig Latin

```
Raw_data = LOAD "sales.rdf" as (Subject, Predicate ,Object);
```

```
Join_res = Join Raw_data by Object, Raw_data by Subject
```

```
Res = FILTER Join_res By $1 eq "Bought" AND $4 = "location" AND $5 =  
"NC";
```

Output = FOREACH Res GENERATE (\$0, \$1, \$5);

In Example 4.1 we perform LOAD, JOIN, FILTER and FOREACH operations. Each of these operations requires reading the data file once completely. Thus the above query collectively reads the data file five times resulting in cost inefficient query execution. Further, the complex queries require executing multiple group operations with the different aggregation functions resulting in more expensive query executions.

One alternative is to implement the complex data processing using the UDF that allows the users to implement the desired functionality as a user defined function. However, in our earlier discussions we have mentioned the disadvantages of this approach. This section presents an extension to the Pig Latin language that includes the specialized functions that allow the complex data processing tasks to be specified in terms of the MD-join operator. Also, the additional classes of expressions are introduced in the language to deal with the graph structured nature of the RDF data.

In the following sub sections, we define the three functions that can be used in the complex ad-hoc data analysis. We first define the *path expressions* to access the related RDF tuples. The path expression can be of two types, the

Class expressions or the *Property expressions*. Class expressions are represented by type : class_name, and are used to specify the class of the subject in the qualifying triple. For example, the expression “type:Customer” specifies that the qualifying tuple’s subject will be of the type Customer. The properties are represented similarly using the property expressions. The graph based nature of the RDF data makes it necessary to specify the navigational patterns of a set of the desired objects that can be represented using property expressions. For instance, the path expression to represent the navigation from the *Customer C1* to the *product P1’s price* can be represented as “bought.price”. These kinds of expressions represent the relation between the tuples and are hence useful to in performing the JOIN operation between the related tuples.

4.1 Generating Fact Dataset: GFD

MD-join operation requires a fact dataset and a base dataset to execute the algorithm. In order to generate the fact dataset, we need to load the RDF file initially. As mentioned earlier, the format of an RDF file is of the form <Subject, Property, Object>, which differs from the format of the relational data (sequence of tuples). Furthermore, the RDF data is accompanied by its metadata in an

input file and must be handled during the load process. Thus we call the LOAD operator of Pig Latin along with the GFD operator. To generate the fact dataset from an RDF file “*input.rdf*”, a specialized class for GFD needs to be added to the Pig Latin library. The following shows the syntax of the GFD function:

```
fact_dataset = LOAD 'input.rdf' USING  
  
GFD(Class_Expression; property_expressions;  
  
aggregation_pathexpression; filter_pathexpression');
```

In this syntax, *input_dataset* is the data loaded from the RDF file. *Class_Expression* indicates the value of the subject in the *input_dataset*. *property_expressions* indicates the properties for which the aggregation needs to be computed. The *filter_pathexpression* indicates the properties for which the filter conditions needs to be checked. Finally, the *aggregation_pathexpression* holds the property on which the aggregation operation is performed. The LOAD operator reads each line from the *input.rdf* file and calls the GFD operator. The GFD operator groups together the tuples based on the subject value and the

necessary JOIN operations are performed to reassemble the tuples. Generating the Fact tuples for the example 4.1 is as shown below. Figure 8 shows the steps in executing GFD for the example given.

```
fact_dataset = LOAD 'input.rdf' USING
GFD (TYPE:CUSTOMER; BOUGHT.LOC,BOUGHT.PRICE;
      BOUGHT.PRICE; BOUGHT.LOC' );
```

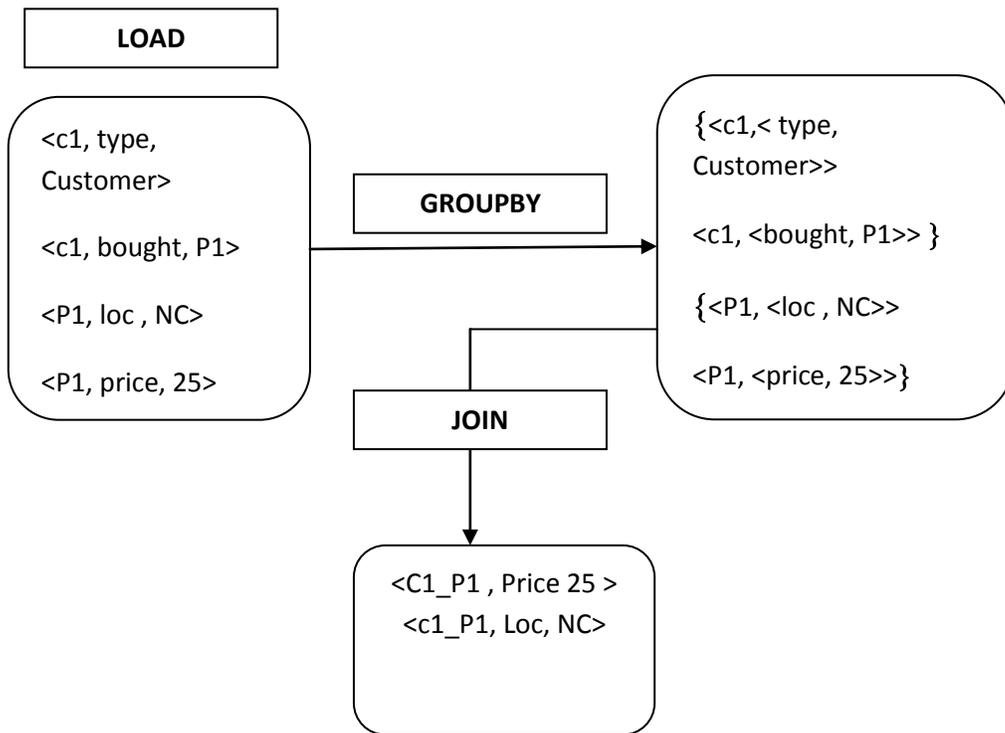


Figure 8: Execution on GFD for Example 4.1

GFD performs the required join operations on the related tuples and generates result of the form <Subject, Property, Object>. The subset of the output for the above query is of the form:

Table 10: Shows the result obtained after executing GFD

Subject	Property	Object
C1_P1	Price	25
C1_P1	Loc	NC
C1_P2	Price	35
C1_P2	Loc	NC

The result generated using this operator is called as the *fact dataset*. Fact dataset is a subset of the tuples that are required to compute the result for the user given query. Within the GFD function, we call the STORE function to store this fact dataset in an intermediate file called the *MDJ.rdf*. The data from the file is later used by the MDJ operator while performing the MD-join operation, which is discussed in section 4.3.

4.2 Generating Base Dataset: GBD

Section 2.1 describes a simple algorithm for a MD-Join operator. The algorithm requires a set of container tuples of all the combinations of the properties for

which the aggregation needs to be computed. For every tuple in the fact dataset, the corresponding combination in the base dataset is obtained and the aggregation results are updated in the container tuples. Similar to the GFD, the GBD operator is executed along with the LOAD function.

```
base_dataset = LOAD 'input.rdf' USING  
  
GBD(Class_Expression; property_expressions; FLAG');
```

As in GFD, the *class_expression* and the *property_expression* are path expression to indicate the relationship that exists between the tuples having the same subjects. The Flag holds either the value “NULL” or “BOTH” that indicates that the key for the Aggregation is either the properties got from the *property_expressions* or a combination of the *property value* and the *type class*. The tuples generated by the GBD are of the type <Subject, Base, NULL> where “Base” is a flag that indicates that the tuple belongs to the base dataset. The NULL value will be replaced by the value computed by the aggregation function when executing MDJ operation. Generating the Fact tuples for the example 4.1 is as shown below. Figure 9 shows the steps in executing GFD for the example given.

```
base_dataset = LOAD 'input.rdf' USING GBD
```

(TYPE:CUSTOMER; BOUGHT.LOC,BOUGHT.PRICE; NULL);

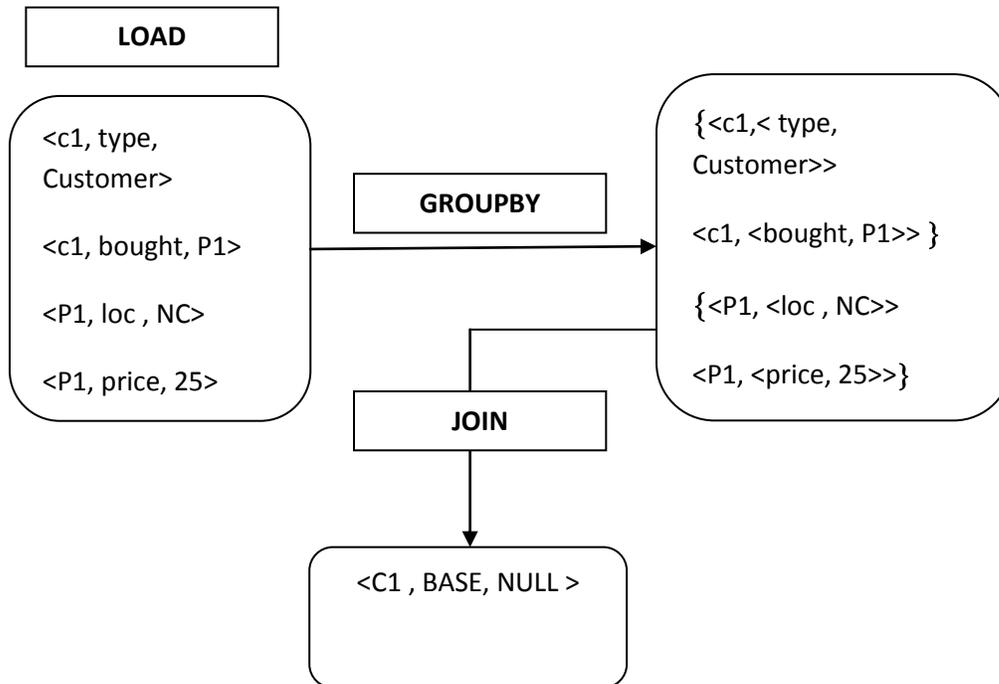


Figure 9: Execution of the GBD for the Example 4.1

The result generated after the execution of GBD is shown in the Table 4

Table 11 : Subset of the result generated after the GBD operation

Subject	Property	Object
C1	BASE	NULL
C2	BASE	NULL

The tuples generated by the GBD operator are referred to as the *base tuples*. Base tuples are initialized with a NULL, for each object corresponding to the subject. The NULL values are updated during the aggregation operation. Within the GBD function, we call the STORE function to append the base dataset into the same *MDJ.rdf* file. This file is later loaded by the MDJ operator while performing the JOIN operation and is discussed in the next section.

4.3 Multi-Dimensional Join: MDJ

After the generation of the base tuple and the fact tuple sets, the next step is the execution of the MD-Join algorithm on these datasets. In order to perform the multi dimensional JOIN operations in the Pig Latin, the MDJ operator class is included as a part of the language library. The MDJ operator executes on the data present in the “*MDJ.rdf*” file created by the GFD and GBD operators as mentioned in section 4.1 and 4.2. This operator takes as input the filter condition on which the aggregation needs to be computed and the aggregation function such as the SUM, COUNT, MAX, MIN, AVG. The syntax for the MDJ operator is as follows:

```
output_dataset = LOAD “MDJ.rdf” USING MDJ(KEY_NAME;
```

AGGREGATION_FUNCTION : AGGREGATION_PROPERTY;

FILTER_PROPERTY:FILTER_CONDITION);

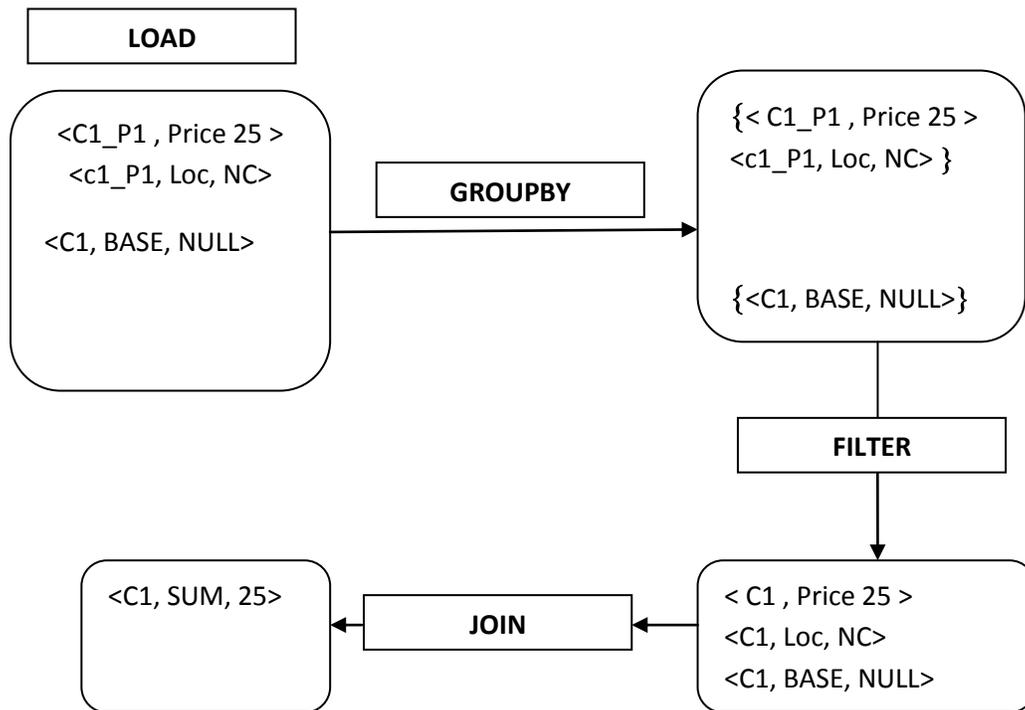


Figure 10: Execution of MDJ for Example 4.1

Executing MDJ operation for the example 4.1 is as shown below. Figure 10

Figure 8 shows the steps in executing GFD for the example given.

output_dataset = **LOAD** "MDJ.rdf" USING **MDJ**(CUSTOMER;

SUM : PRICE; STATE:NC);

The result generated for the above query is as shown in Table 12.

Table 12 : The results generated after the MDJOIN operation

Customer	SUM (PRICE)
C1	25
C2	35

The output generated is stored in an output file using the Pig Latin primitive the STORE function.

Using the above mentioned operators- GFD, GBD and MDJ; it is possible to perform various kinds of analytical queries. The following shows examples of executing simple queries, CUBE BY, pivoting, GROUP BY operations using the extended Pig Latin primitives.

Example: 4.1 – simple query execution using extended operators

For the Sales data shown in the Chapter 1, suppose we want to compute a simple query to find the total sale in the month of December for each state.

The query for the about scenario has to be of the type, for every possible state present in the fact dataset we need to get the sale for the month December.

The above query when executed using primitive Pig Latin operators, it requires two JOIN operations and one GROUP BY operation along with two FOREACH iterations to perform the reassembling of the data and computing the aggregation. Each of these operations will require reading the dataset completely. Using the extended Pig Latin primitives we can represent the above examples as:

Assuming the data is the “*sales.rdf*” file we can execute GFD to reassemble the RDF tuples in a way that it groups together all the related tuples. GFD can be executed as:

```
fact_dataset = LOAD 'sales.rdf' USING  
  
GFD (TYPE:CUSTOMER; BOUGHT.LOC;BOUGHT.PRICE;  
  
BOUGHT.MONTH) ;
```

Bought.LOC is the property for which we need to compute the sales. The aggregation is computed on the BOUGHT.PRICE property and the filter condition is checked on BOUGHT.MONTH property.

After the creation of the fact table, we need to create the base data using the GBD:

```
base_dataset = GBD 'sales.rdf' USING  
  
(TYPE:CUSTOMER; BOUGHT.STATE; NULL);
```

The above query generates the base dataset containing tuples for every state present in the 'sales.rdf'. Every record in the base dataset has its property set to "BASE" and the value of the Object set to "NULL". After the generation of the fact and the base dataset, MDJ operator can be executed on these datasets

```
output_dataset = LOAD "MDJ.rdf" USING MDJ(STATE;  
  
SUM : SALE; MONTH:December);
```

The above query reads every tuple from the fact dataset and iterates over the base dataset to find a tuple where the State of the fact data and the base data is the same and the month in the fact data is equal to December. When a corresponding match is found, the base dataset is updated with the sum of the

sale. The result is thus obtained by performing one join for each of the GFD, GBD and MDJ operations, thus reducing the number of scans over the dataset.

Example 4.2: Pivoting example

Compute the average purchase for each customer in the month Jan, May and Dec.

Executing the above example using Pig Latin language primitives requires reading the dataset multiple times to perform the GROUP, JOIN and FOREACH operations. These operations are required to compute the aggregation over multiple dimensions. To execute the same using extended Pig Latin operators, we execute GFD using *CUSTOMER* class as the base class and the aggregation is computed on the property *PRICE*. The filter condition is checked on the property *MONTH*. The query for the example is as shown below:

```
fact_dataset = LOAD sales.rdf USING  
  
GFD(TYPE:CUSTOMER; NULL,  
  
BOUGHT.PRICE; BOUGHT.MONTH);
```

Step 2: To execute the MDJ operation it is required to generate the base dataset. Base tuples are generated for each combination of the properties for which the aggregation is computed. The predicate for these tuples are set to BASE and the object has the value set to NULL initially, which will be updated during the MDJ operation. The GBD for generating the base dataset for the given query is as shown below:

```
base_dataset = LOAD sales.rdf' USING  
  
GBD(TYPE:CUSTOMER; NULL ; NULL');
```

Step 3: After the base and the fact dataset is generated, we need to compute the MDJ operation. For the given query, we need to compute average price for the month- Jan, May and Dec. Hence we execute three queries to compute MDJ one for each month specified.

```
initial_dataset = LOAD "MDJ.rdf" USING  
  
MDJ(CUSTOMER; AVG :PRICE; MONTH:JAN);
```

Step 4: MDJ with filter condition Month = May

```
initial_dataset = LOAD "MDJ.rdf" USING
```

```
MDJ(CUSTOMER; AVG :PRICE; MONTH:MAY);
```

Step 5: MDJ with filter condition Month = Dec

```
output_dataset = LOAD "MDJ.rdf" USING
```

```
MDJ(CUSTOMER; AVG :PRICE; MONTH:DEC);
```

The output_dataset contains the sales of each customer who purchased products in the month of Jan, May and Dec respectively.

Example 4.3: Data cube example

To compute the number of sales above the average sale, when we are viewing the Sales from all possible combinations of Prod purchased, Month when the purchase was made and the state where the purchase was made.

In order to execute these queries using primitive operators after the reassembling of the data, we need to perform, eight group by operations for all possible combinations of attributes .i.e., none, prod, month, state, prod & month, prod & state, month & state , prod & month & state. Each group by operation computes the average sale for the possible combination of attributes. We require eight more subqueries, where each subquery performs a join

operation with the original sales dataset to validate the filter condition i.e. $\text{Avg}(\text{sale}) > \text{sale}$. Finally to get the count of the number of sales above the average sale, we perform eight group by operations with the same combination of attributes as the first set of group by operations. This provides the results where the count of the sales is above the average sale for various combinations of attributes.

The solution of this kind will not result in an efficient query execution due to the execution of 16 group by operations and 8 join operations, where each of these operations require one complete scan of the dataset, which is an expensive operation.

A cost efficient alternative for the above query can be obtained by executing the above using Pig Latin extended primitive as shown below:

Step 1: For the given query, fact dataset is generated using the GFD operator.

The query is as shown below

```
fact_dataset = LOAD sales.rdf' USING  
  
GFD(TYPE:CUSTOMER;  
BOUGHT.STATE,BOUGHT.MONTH,BOUGHT.PROD;
```

```
BOUGHT.PRICE; BOUGHT.PRICE);
```

Step 2: Base dataset for the given query contains tuples with subject having all possible combinations of the properties – state, month and prod. The query to generate base dataset for the given examples is as shown below:

```
base_dataset = LOAD salse.rdf' USING
```

```
GBD(TYPE:CUSTOMER;
```

```
BOUGHT.STATE,BOUGHT.MONTH,BOUGHT.PROD;
```

```
NULL' ) ;
```

Step 3: Query to perform MDJ that computes the average sale is as shown below:

```
output_dataset = LOAD "MDJ.rdf" USING
```

```
MDJ(STATE,MONTH,PROD; COUNT : PRICE; PRICE > AVG_PRICE);
```

In step 1, reassembling of the related tuple is performed and the result of GFD will provide the required fact data set to perform the multi dimensional join operation. Step 2 creates the container tuples, where the aggregation values

can be updated when executing the MDJ operation. Finally step 3 and 4 perform the actual MDJ operations. In step 3 we compute the average sale for all the combinations of attributes. Using this average, in step 4 we compute the number of sales above the average sales for the all the combinations of *PROD*, *MONTH* and *STATE*.

Chapter 5

Implementation

The queries written in the Pig Latin are executed as the Map-Reduce jobs using the Hadoop system. The new architecture of the Pig Latin system after including the extended operators is as shown in Figure 7

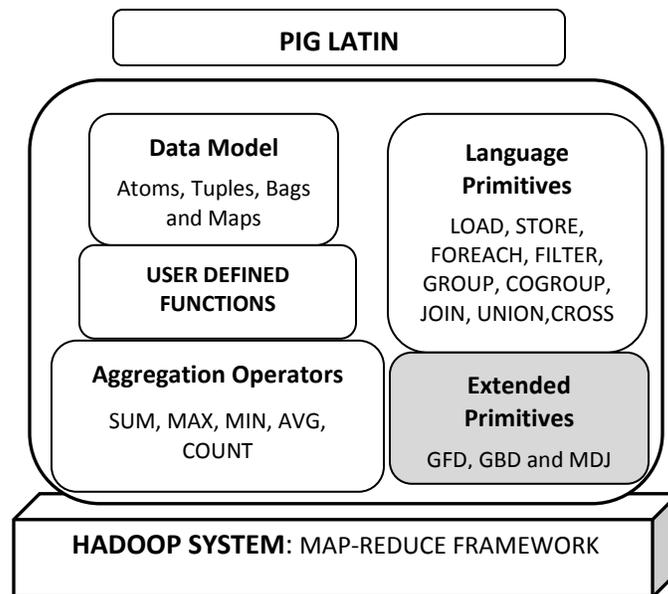


Figure 11: Architecture of Pig Latin Language

5.1 Execution plan on Hadoop

The operators GFD, GBD and MDJ execute on the Map-Reduce framework. The Map-Reduce framework is designed in such a way that it divides the huge tasks into the smaller tasks and assigns them to every processor for processing. The GFD and the GBD operators group all the tuples based on the *Subject* and then perform the multiple join operations on them to reassemble all the related tuples. Each grouping operation is compiled into one Map-Reduce dataflow, where every tuple is assigned with one key in such a way that all the tuples with the same subject will have the same key. The Reduce function will group together all the tuples with a similar key which are required to perform the join operation. Furthermore, to reduce the relations between these tuples we will require a Map-Reduce dataflow for each of the join operations. Hence the number of Map-Reduce iterations depends on the number of join operations that will be performed to reassemble the records. Figure 8 shows the execution plan of GBD and GFD on Map-Reduce

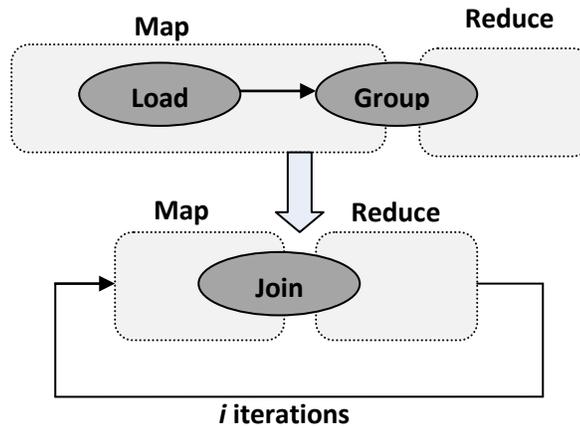


Figure 12 : Execution plan of GBD and GFD on Map-Reduce

The MDJ gets compiled into one Map-Reduce workflow, where the properties on which the aggregation needs to be computed are grouped together in the Map phase and the actual aggregation is performed in the Reduce phase. Figure 5 shows the execution of MD-join on Map-Reduce.

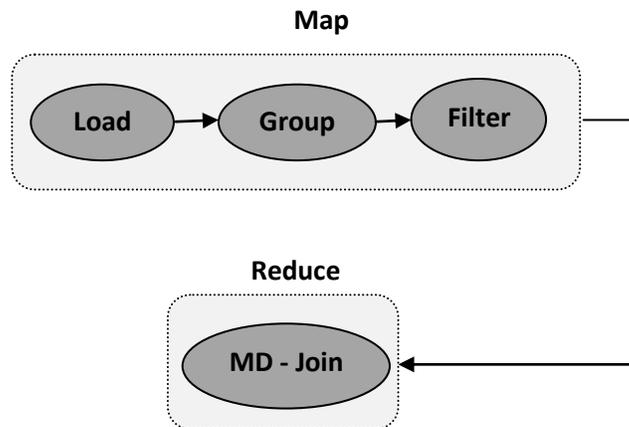


Figure 13 : Execution plan for MDJ on Map-Reduce

In the Pig Latin language, the *execution plan* and the *execution platform* are dependent on each other. The execution platform for the queries is the Hadoop system. In this section, we have seen the *execution plan* for the extended operators that we have implemented for the Pig Latin language.

Chapter 6

Evaluation

6.1 Environment

For the evaluation purpose, queries of different complexities were executed on a Hadoop setup using an actual dataset and a synthetic dataset, primary created to evaluate the performance of SPARQL queries. Two Hadoop instances are created on the *Virtual Computing Lab (VCL)* machines one as the *Job Tracker* and the other as the *Task Tracker*. One instance of the Job Tracker is tagged with three instances of the Task Tracker, thus creation of one reservation for the Job Tracker creates one master and three slave processors. Hadoop is installed on Red Hat Enterprise Linux and Java 5 is used for the execution of the code. The following subsections show few examples of the

queries and their cost evaluation. In these examples, we have used short hand representations of the URI's for the sake of readability.

6.2 BSBM dataset

We conducted a cost performance evaluation based on a dataset from *Berlin SPARQL Benchmark (BSBM)* [5]. BSBM is a synthetic dataset for evaluating the performance of the SPARQL queries. In this dataset, the information about the *Vendor*, the *Offers* provided by the *vendors* on the various *Product types* and the relationships between the *offers* and the *products* are mentioned. We consider a subset of that graph and the relationship between the nodes for our evaluation. Figure 14 represents the subset of the BSBM schema.

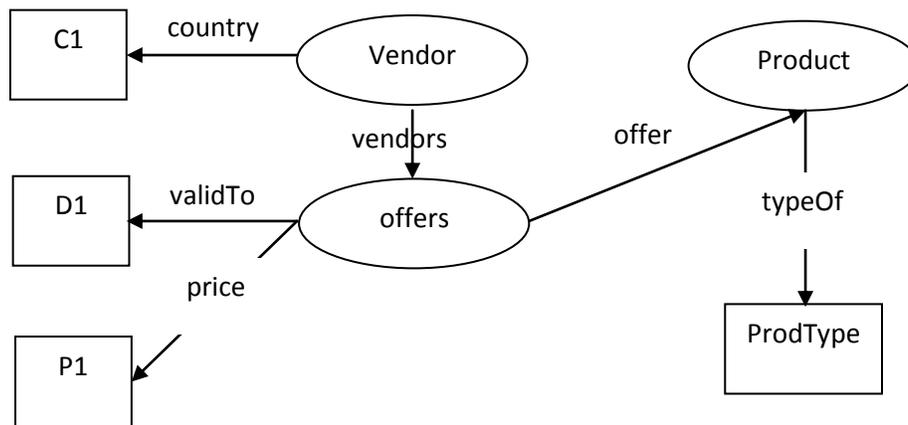


Figure 14: Example subgraph taken from the BSBM dataset

6.2.1 Query Execution on BSBM dataset

Consider an analytical query, to compute the number of offers made by the *vendors*, who were above the average price, when the data is viewed from all the combinations of *typeOf*, *validTo* and *country*. In order to compute the result of the given query, we need to initially reassemble the RDF data in order to get the offer price related to each offer made by the vendor. We assume that the RDF data for the BSBM dataset is in the *input.rdf* file. The data can be reassembled by performing the GFD operation as shown below:

```
fact_dataset = LOAD 'input.rdf' USING  
  
GFD(TYPE:VENDOR; VENDORS.VALIDTO, VENDORS.COUNTRY,  
  
VENDORS.OFFERS.TYPEOF; VENDORS.PRICE;  
  
VENDORS.OFFERS.TYPEOF');
```

Next, in order to execute the MDJ operation we need to have the base tuples, which are generated using the GBD operator as shown below –

```

base_dataset = LOAD 'input.rdf' USING

GBD(TYPE:VENDOR; VENDORS.VALIDTO, VENDORS.COUNTRY,

      VENDORS.OFFERS.TYPEOF; NULL');

```

The MDJ operator can now be executed using the base_dataset and the fact_dataset. MDJ operator has to be executed initially to compute the *average price* of the *offers*. After the computation of the average price, the MDJ operator has to be executed to compute the *Count* based on the average price calculated. The MDJ operator for the two Aggregations is as shown below –

```

output_dataset = LOAD "MDJ.rdf" USING

MDJ(TYPEOF,VALIDTO, COUNTRY; AVERAGE : PRICE; NULL);

final_dataset = LOAD "MDJ.rdf" USING

MDJ(TYPEOF,VALIDTO, COUNTRY; COUNT :

      PRICE; PRICE > AVG_PRICE);

```

6.2.2 Results

We compare the costs between the execution of the above query using the Pig Latin primitive operators and the new extended operators. Table 13 shows the cost analysis for the query execution at various steps.

Table 13: Cost analysis for the query execution on BSN dataset

	Pig Latin Primitive operators	Pig Latin Extended operators
Number of User queries	27	4
Number of Joins required to reassemble the data	3	3 – GFD 3 - GBD

Table 14: continued

Number of GroupBy operations required	8 (Before we perform the Join operation)	1 – GFD
	8 (After the Join operation)	1 - GBD
Number of Joins required after the reassembling of the data	8 Joins	2 – MD-Joins
Execution Time (File Size of 3.6MB)	16.23 Minutes	6.37 Minutes

Table 13 shows the cost analysis, for executing the user query in Pig Latin using the primitive operators and the proposed extended operators. It can be

clearly seen that the number of user written queries using the extended operators is very less when compared with the queries written using the primitive operators. The number of Join operations for reassembling the tuples using extended Pig Latin operator is slightly more than the number of Joins when executing the same operation using primitive language operators. The reason being, to perform the MDJ operation, we need the base dataset and the fact dataset, resulting in increased number of Join operations. Even though the cost incurred when reassembling the tuple is more in the case of the extended operators, the cost for performing the multi-dimensional join is one fourth of the cost for performing the same operation using the primitive operators of Pig Latin language. Thus this approach seems more cost efficient than using the primitive operators. Figure 8 shows how the cost on extended operators decreases after the reassembling the data.

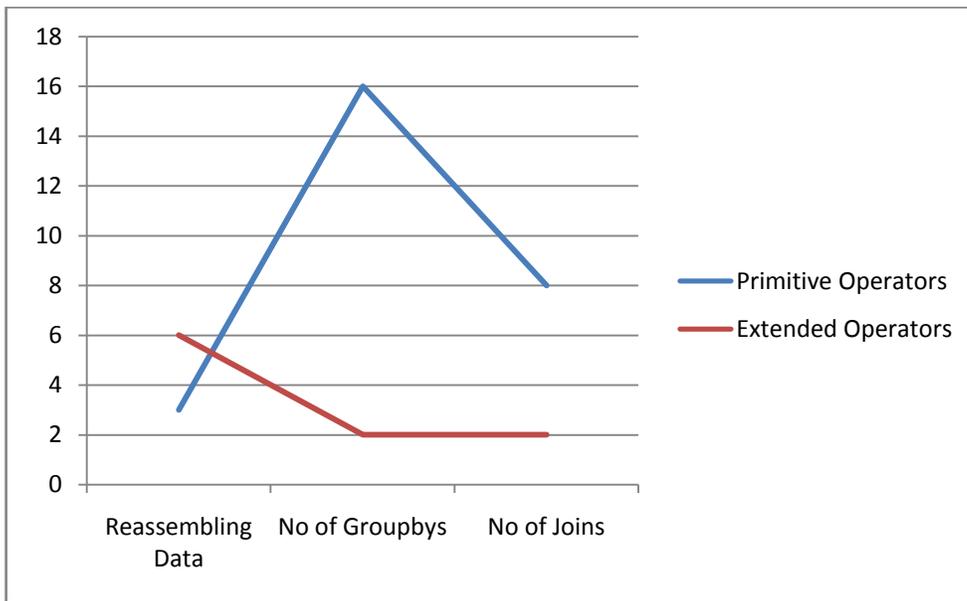


Figure 15: Graph shows the cost analysis using the two approaches

6.3 DBLP dataset

DataBase systems and Logic Programming (DBLP) dataset is the RDF representation of DBLP publication portal which is part of the SWETO dataset [25]. It contains the information that represents the relationship between the *Authors of the Books, the book publications, Year of Publication* and other related nodes. Figure 16 shows the subset of the DBLP schema.

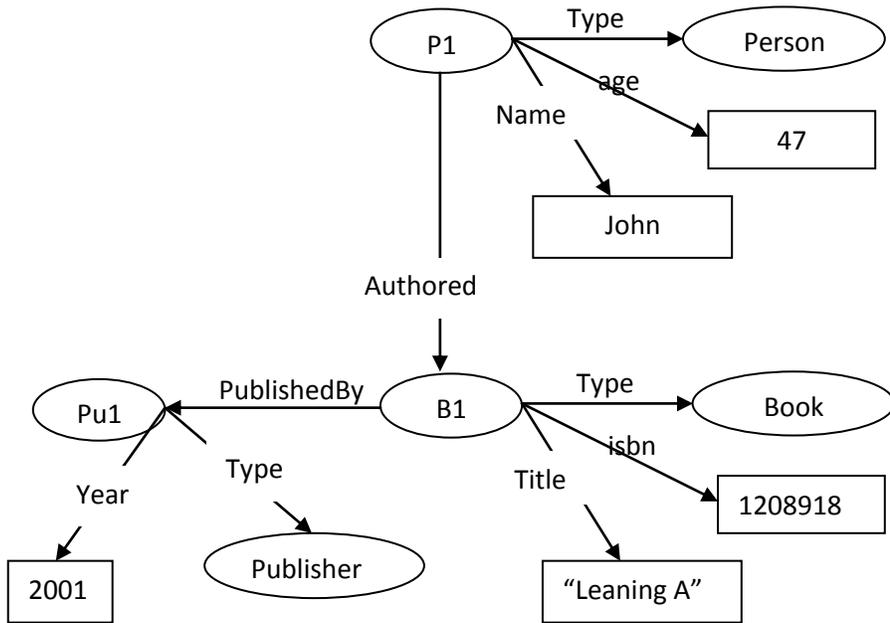


Figure 16: Example subgraph taken from the DBLP dataset

6.3.1 Query Execution on DBLP dataset

Consider a query, to compute the number of books written by each author in the years “1999”, “2003” and “2007”. To compute this result, let us assume the RDF data for that the DBLP dataset is in the *input.rdf* file. The first step is to reassemble the data using the GFD operator which is as shown below:

```
fact_dataset = LOAD 'input.rdf' USING
```

```
GFD(TYPE:PERSON; AUTHORED.ISBN,
```

```
AUTHORED.PUBLISHEDBY.YEAR;
```

```
AUTHORED.ISBN; AUTHORED.PUBLISHEDBY.YEAR);
```

The base dataset for the above data is generated using the GBD operator. The query below shows the generation of the base dataset.

```
base_dataset = LOAD 'input.rdf' USING
```

```
GBD(TYPE:PERSON; NULL ; NULL');
```

Since we need to compute the Aggregation for every year and for each author, we execute the MDJ once for every year. Hence we need three MDJ operations to get the expected result.

```
initial_dataset = LOAD "MDJ.rdf" USING
```

```
MDJ(PERSON,ISBN; COUNT : ISBN; YEAR:1999);
```

```
output_dataset = LOAD "MDJ.rdf" USING
```

```
MDJ(PERSON,ISBN; COUNT : ISBN; YEAR:2003);
```

```
final_dataset = LOAD "MDJ.rdf" USING
```

MDJ(PERSON,ISBN; COUNT : ISBN; YEAR:2007);

The output of the *final_dataset* contains the count of the number of books written by each author in the years “1999”, “2003” and “2007”.

6.3.2 Results

Table 15 shows the cost analysis for the above query and compares the cost of executing this query in the Pig Latin language using the primitive operators with the cost of executing the same query using the extended Pig Latin operators.

Table 15: Cost analysis for the query execution on DBLP dataset

	Pig Latin Primitive operators	Pig Latin Extended operators
Number of User queries	11	5
Number of Joins required to reassemble the data	3	3 – GFD 3 - GBD

Table 16: Continued

Number of GroupBy operations required	6	1 – GFD 1 - GBD
Number of Joins required after the reassembling of the data	4 – Joins + 4 – Joins	3 - MDJ
Execution Time (File Size of 6.6MB)	38.08 Minutes	18.19 Minutes

Table 15 shows the cost evaluation for the user query using the two approaches. In this case we see results similar to the results seen in Table 13. Initially the cost of reassembling the data is more when executing the queries using the extended operators, due to the creation of the fact and the base dataset. After the reassembling of the data, the query execution cost reduces to close to one fourth of the cost for executing the same query using the primitive

operators of the Pig Latin language. Figure 10 shows, how the cost for the query execution decreases after reassembling the data in the case of the extended Pig Latin operators.

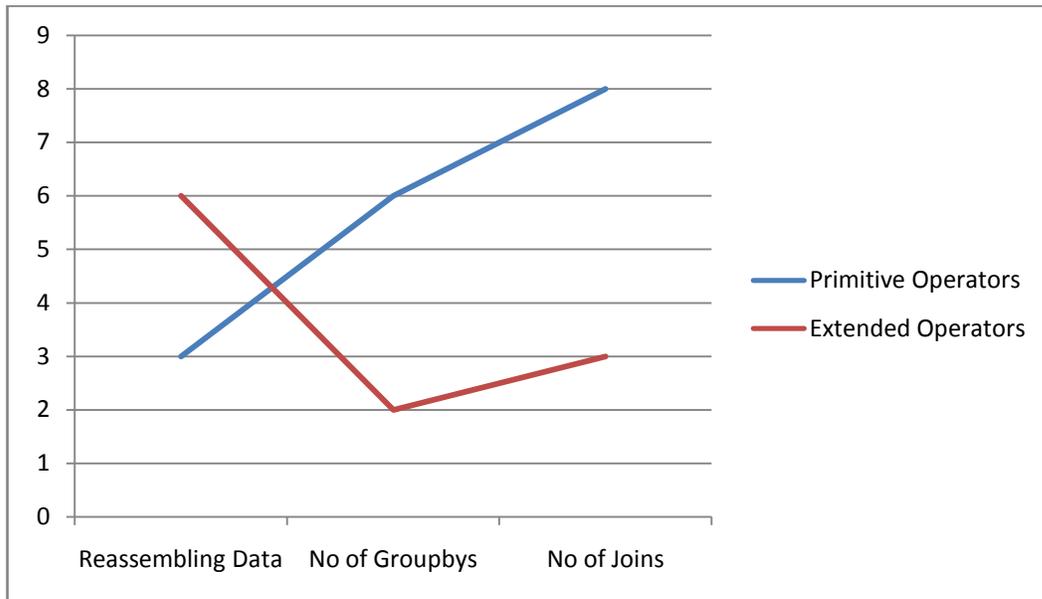


Figure 17: Graph shows the cost analysis using the two approaches

In Table 15 and Table 16, the first row indicates the number of user queries that are required to perform the required complex analytical analysis. As the result shows, the number of queries required to perform the task using the primitive operators is very large when compared with the number of queries written using

the extended operators to perform the same task. This result demonstrates another advantage using the extended operators.

Chapter 7

Related Work

We have provided a set of operators to perform MD-join operation in Pig Latin language. These operators can be used to perform complex analytic querying on RDF datasets. In this section, we compare this approach with other data processing languages and OLAP techniques

Analytical queries are often very complex requiring multiple aggregations over multiple groupings. Such queries are cumbersome to express using traditional query operators such as the GROUPBY and often lead to inefficient query plans. OLAP operators such as CUBE, ROLLUP are relatively recent additions to the SQL allow for more succinct expression of a subset of analytical queries. However, many of the complex ad-hoc analytical queries still remain a

challenge to express using these operators. The MD-Join achieves succinct and efficient expression of a broader range of analytical queries by decoupling the grouping and the aggregation functions.

Parallelization is an increasingly popular method of achieving scalable processing on large datasets. High end database systems with specialized parallel architectures are one alternative for supporting efficient complex OLAP query processing. Various parallel database products like Teradata, Tandem, NCR, Oracle-n CUBE, and RAC are designed to provide impressive scalability and efficient query processing speed. However, are very expensive and are not designed to scale to the size of data on the Web. Computational clusters of commodity grade machines provide a cost effective alternative to high end parallel database systems that can be used to achieve the kind of scalability needed for performing analytical querying on the Web data. Map-Reduce framework provides a simple yet efficient way to enable parallel processing on such clusters without requiring the user to understand the complexities of distributed systems. The map and reduce functions are mainly suitable for computations that are conceptually straightforward and as the problem grows in complexity, the optimization is limited by the custom code written by the user. Task-specification languages provide a way to overcome this opaque nature of

Map-Reduce framework. Languages like Yahoo's Pig Latin [6], Google's Sawzall [24] and Microsoft's DyradLINQ [10] are high-level languages that implement data processing tasks and have been built on top of Map-Reduce.

DyrandLINQ is a high level language built over a distributed platform to provide large scale data processing, over a parallel and fault tolerant execution process. The distributed platform is called Dyrand that is developed by Microsoft [95]. Unlike the rigid two step chain of Map-Reduce, Dyrand provides more flexibility in performing arbitrary computations. But the limitations of this system is that the high level language, DyranLINQ is hard to program and the languages is not widely known to the public.

Sawzall is another language used by Google that executes over the Map-Reduce framework. It is a scripting language with rigid structure similar to the Map-Reduce framework with the filter operations performed in the Map phase and the aggregations performed in the Reduce phase. The language allows user defined functions to be implemented only in the filter phase and has a limited set of pre-defined aggregation operations. This provides limited flexibility in processing complex queries on RDF datasets. While in Pig Latin, the language provides a wide range of primitives for data computation and also allows user defined functions to be implemented. Operators like Group,

CoGroup are useful in perform various join operations that are required when processing analytical queries on RDF datasets.

Pig Latin is a balance between a high-level declarative constructs of SQL and a low-level procedural way of Map-Reduce. This provides opportunities to implement complex operators that can be scalable and thus can process the data at the Web scale efficiently. Map-Reduce platform is designed to provide fault tolerance, store data locally to avoid network bandwidth issues and also maintain backup tasks for faster and reliable execution.

Chapter 8

Future Work

A data flow language like the Pig Latin provides the required flexibility to the users and is suitable for the simple data processing tasks. In this report we have shown how to implement the MD-join operator in the Pig Latin language for the analytical processing of the RDF datasets. There are many promising areas that are yet to be explored in the context of the analytical processing of the RDF datasets using a scalable approach like the Pig Latin Language. The following are a few possible research areas.

➤ Optimizations:

In this section we examine how we can optimize these extended operators.

- a) *Generalizing MD-Joins*: Currently, for a certain complex MD-join operations, we perform the nested MD-join executions. Various ways to

perform the generalized multi-dimensional joins can be researched, resulting in an efficient execution, as it reduces the number of Map-Reduce cycles considerably. The focus of this research should be to identify, develop and implement a set of primitives that will be required to express the queries in terms of a generalized multidimensional join. Doing so will enable us to overcome the shortcomings of the traditional operators and enable highly efficient Map-Reduce workflows.

- b) *Indexing the datasets*: In order, to reassemble the data to perform the complex join operations, we iterate multiple times through the existing set of the user input data to find the related tuples. By finding an appropriate way to index these data, multiple iterations on the same dataset can be avoided.
- c) *Query rewriting*: Rewriting the rules for the optimization of the queries can be further researched. Such rules could include the transformation of the queries that use the traditional operator to perform the multi-dimensional querying to MD-join queries. Further, any of the nested MD-join queries must be identified and transformed into generalized multi dimensional queries.

- User interfaces: The productivity of the framework can be enhanced through a right interactive interface. Currently the queries are executed on a command line interface, which has limited support for validating the user inputs. A well designed user interface that provides options to validate the user entered input will avoid unnecessary query executions.

- Providing keyword notations instead of lengthy URIs: The current framework takes the entire URI to represent the path expressions. An efficient approach might be to take the keywords as the input from the user and to map them to the actual URI's of the RDF triples during the computation of the Aggregations. This approach makes the language user friendly and increases the productivity.

Chapter 9

Conclusion

In this work, we presented an approach for scalable analytical processing on the RDF datasets using the parallel processing techniques. This approach extends on the existing platforms such as Hadoop, Map-Reduce, Pig to provide the structure and the semantics of the RDF data. Further, we integrated the multi-dimensional join operator to perform the analytical processing on the graph structured data like the RDF. We provided a simple and intuitive syntactic extension of Pig Latin language in order to express the MD-join on the RDF dataset. We also demonstrated the usability of this approach using the case studies on the synthetic and the real datasets and tabulated the obtained results.

REFERENCES

- [1] ACM SIG PROCEEDINGS template.
<http://www.acm.org/sigs/pubs/proceed/template.html>.
- [2] Apache Incubator Projects Proceedings <http://incubator.apache.org/pig/>
- [3] Apache Projects Proceedings <http://hadoop.apache.org/core/>
- [4] ARQ Proceedings <http://jena.sourceforge.net/ARQ/>
- [5] BSBM Proceedings
<http://www4.wiwiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/spec/index.html#dataschema>
- [6] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar and Andrew Tomkins. “Pig latin: a not-so-foreign language for data processing”. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, p.1099 -1110, 2008
- [7] Damianos Chatziantoniou. “Using grouping variables to express complex decision support queries”. In Proceedings of the 2007 Data & Knowledge Engineering conference, p.114–136, 2007
- [8] Daniel J. Abadi, Adam Marcus, Samuel R. Madden and Kate Hollenbach. “Scalable Semantic Web Data Management Using Vertical Partitioning”. In proceedings of the 33rd international conference on Very large data bases, p. 411 - 422, 2003

- [9] D. Chatziantoniou M. Akinde, T. Johnson, and S. Kim “The MD-join: an operator for Complex OLAP” ICDE 2001, 108–121 ”
- [10] Dryad LINQ Proceedings
<http://research.microsoft.com/research/sv/DryadLINQ/2007>.
- [11] F. Manola and E. Miller. RDF Primer. W3C Recommendation, Feb. 2004.
<http://www.w3.org/TR/rdfprimer/>.
- [12] <http://www.flickr.com/>
- [13] <http://www.google.com/coop/cse/>
- [14] <https://secure.delicious.com>
- [15] <http://www.w3.org/RDF/>
- [16] <http://www.w3.org/TR/rdf-sparql-query/>
- [17] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsias, D.Stott Parket. “Map-Reduce-Merge:Simplified Relational Data Processing on Large Clusters”. In proceedings of the 2007 ACM SIGMOD international conference on Management of data, p. 1029 - 1040, 2007
- [18] Jeffrey Dean and Sanjay Ghemawat. “MapReduce : Simplified Data Processing on Large Clusters”. In proceedings of the Sixth Symposium on Operating System Design and Implementation (OSDI'04), 2004

- [19] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart and Murali Venkatrao "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals". In proceedings of the 12th International Conference on Data Engineering (ICDE'96), p.152, 1996
- [20] Khare, Rohit (January/February 2006). "Microformats: The Next (Small) Thing on the Semantic Web?". IEEE Internet Computing 10 (1): 68–75
- [21] M. Isard et al. Dryad: Distributed data-parallel programs from sequential building blocks. In European Conference on Computer Systems (EuroSys), pages 59 - 72, Lisbon, Portugal, March 21-23 2007.
- [22] M.O. Akinde and M.H. Bohlen. "Generalized MD-Joins: Evaluation and Reduction to SQL". In proceedings of the VLDB 2001 International Workshop on Databases in Telecommunications II, 2209:52 - 67, 2001
- [23] M.O. Akinde and M.H. Bohlen. "Efficient Computation of Subqueries in Complex OLAP". In proceedings of the 19th International Conference on Data Engineering (ICDE'03), p. 163 - 174, 2003
- [24] Rob Pike, Sean Dorward, Robert Griesemer, Sean Quinlan. "Interpreting the Data: Parallel Analysis with Sawzall". In Proceedings of the special Issue on Grids and Worldwide Computing Programming Models and Infrastructure 13:4, pp. 227-298
- [25] Swetodblp Proceedings
<http://lstdis.cs.uga.edu/projects/semdis/swetodblp/>
- [26] <http://swoogle.umbc.edu/>

APPENDIX

1. Environment setup

Step 1: To create an image for Hadoop on the VCL machine, select **Manage Images tab** on the VCL page. In the manage images page, select **Create / Update of an Image** and **Submit** the request.

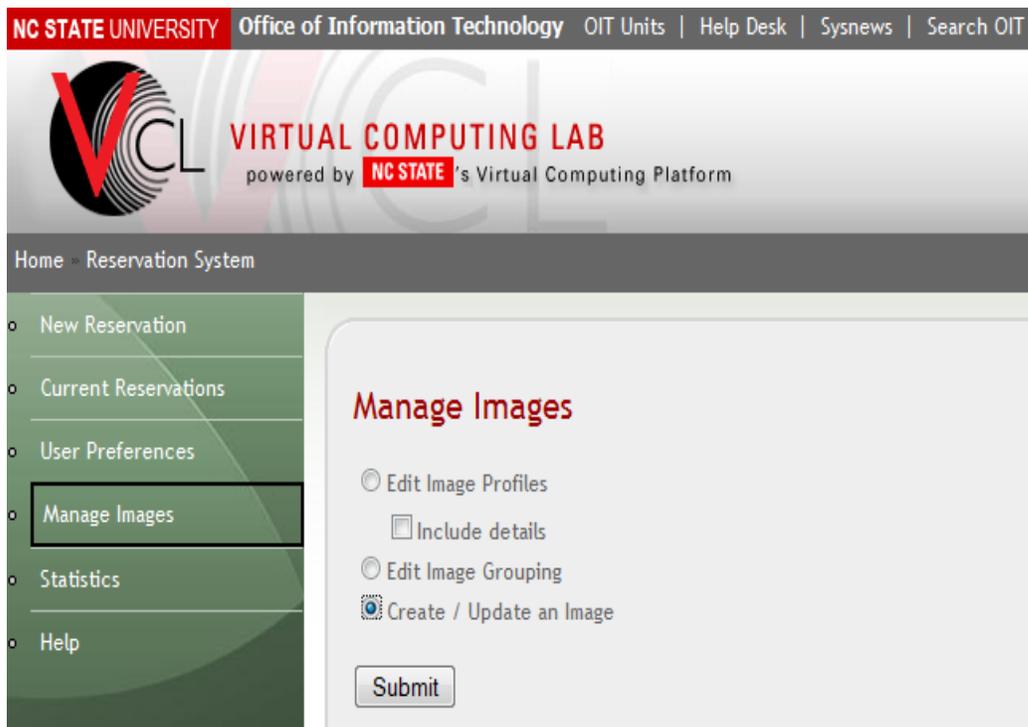


Figure 18: Screen shot of the Manage Images page

Step 2: Hadoop is installed on the Red Hat Linux environment. Hence, in the **Create / Update an Image** page select **Red Hat Enterprise Linux** option from the drop down list. Submit the **Create Imaging Reservation** request.

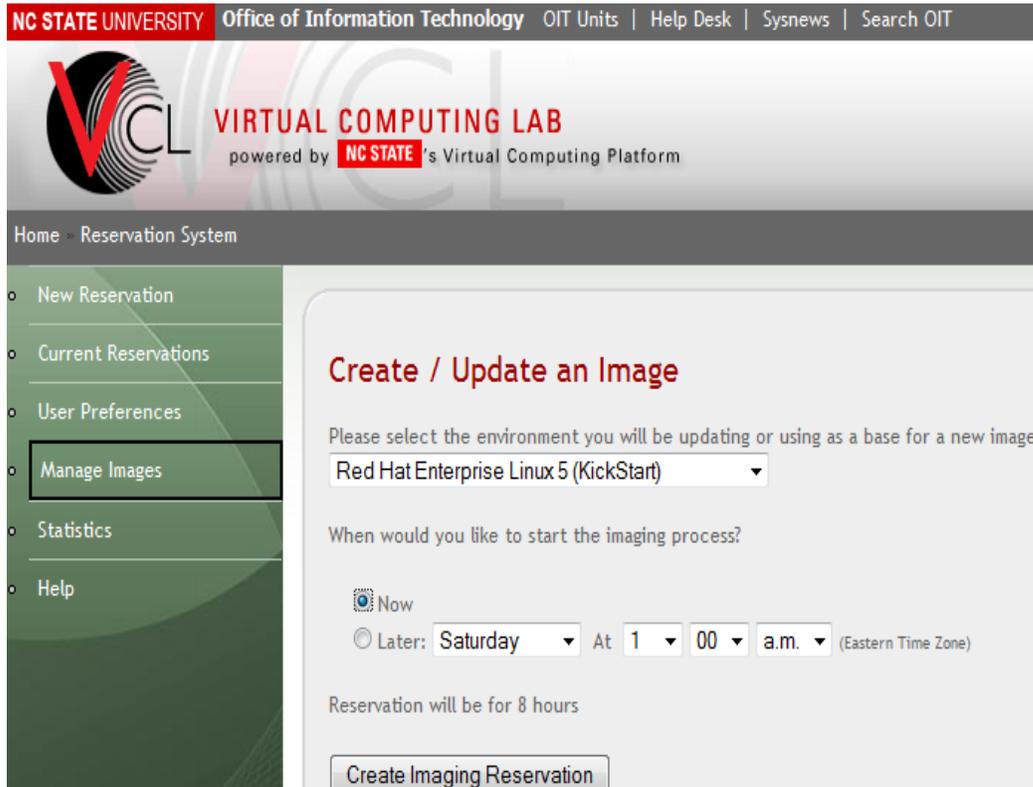


Figure 19: Screen shot of the Create an Image page

Step 3: When the reservation is successfully made, the system provides the remote machine's *IP address* and the *login details* to connect to that system, where the Hadoop image can be created.

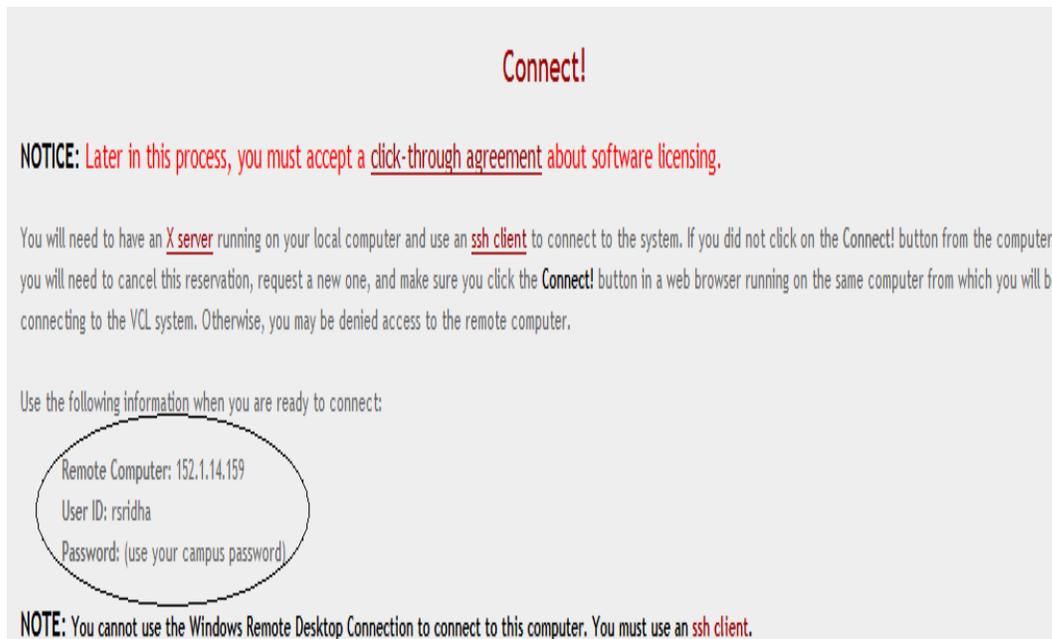


Figure 20: Screen shot of the Connect page

Step 4: Using a **ssh client**, connect to the remote computer. **Hadoop** is installed on this machine. Instance of the Hadoop can be obtained at <http://hadoop.apache.org>. Instructions to install Hadoop can be obtained at: <http://hadoop.apache.org/core/docs/current/quickstart.html>

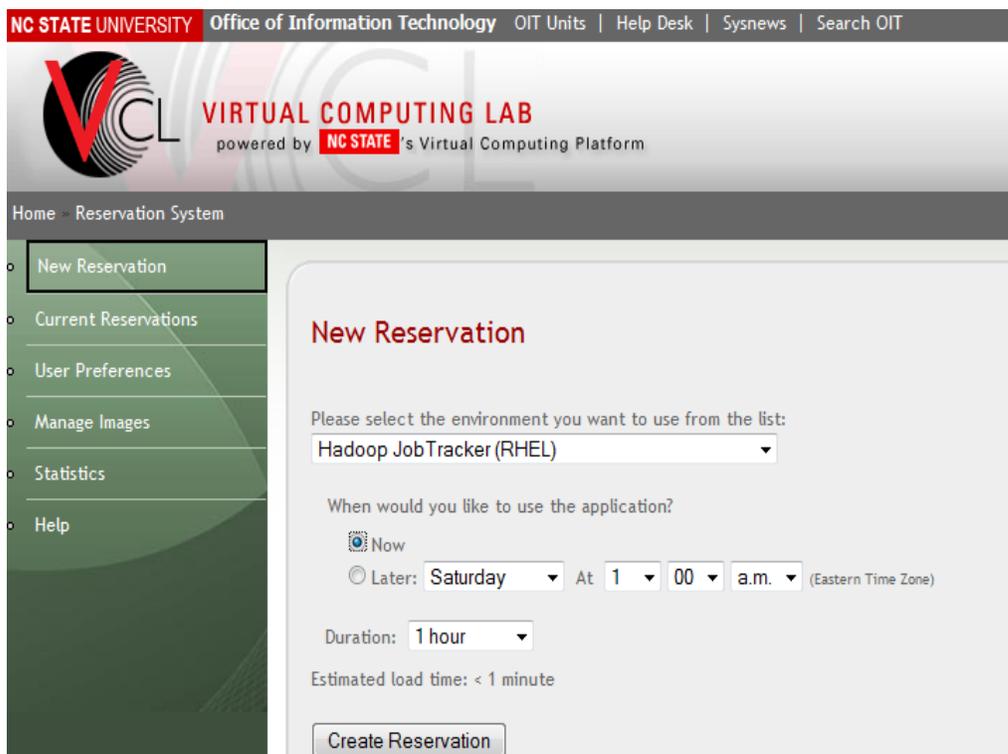
Step 5: For Hadoop execution, we need the *Java environment* setup. *Java 5.x for Linux* software can be downloaded from <http://java.sun.com/>

Step 6: Save this image by naming it the *Job Tracker*. Create another image similar to the Job Tracker and save it as the *Task Tracker*.

Step 7: One Job Tracker is tagged with n number of Task Trackers, where n is the number of the slave process required. The images can be tagged by selecting the **Edit image** option present in the **Manage Images page**.

2. Access to the Hadoop image

Step 1: Make a new reservation, selecting the **Hadoop Job Tracker** environment from the drop down list. And submit the request by selecting **Create Reservation**



The screenshot shows the 'New Reservation' page of the Virtual Computing Lab. The header includes 'NC STATE UNIVERSITY Office of Information Technology' with links for 'OIT Units', 'Help Desk', 'Sysnews', and 'Search OIT'. The logo for 'VIRTUAL COMPUTING LAB' is powered by 'NC STATE's Virtual Computing Platform'. The breadcrumb trail is 'Home > Reservation System'. A left sidebar menu contains: 'New Reservation' (highlighted), 'Current Reservations', 'User Preferences', 'Manage Images', 'Statistics', and 'Help'. The main content area is titled 'New Reservation' and contains the following form elements:

- Text: 'Please select the environment you want to use from the list:'
- Dropdown menu: 'Hadoop JobTracker (RHEL)'
- Text: 'When would you like to use the application?'
- Radio button: 'Now' (selected)
- Radio button: 'Later: Saturday At 1 00 a.m. (Eastern Time Zone)'
- Text: 'Duration: 1 hour'
- Text: 'Estimated load time: < 1 minute'
- Button: 'Create Reservation'

Figure 21: Screen shot of the new Reservation page

Step 2: Based on the number of Task Trackers configured (n), the reservation creates *one Job Tracker (Master)* and *n Task Trackers (slaves)* as configured. The successful creation of the reservation will provide us with a set of *IP addresses* to connect to the master and slave machines.

Connect!

You will need an [ssh client](#) to connect to any unix systems.
You will need a [Remote Desktop program](#) to connect to any windows systems.

Use the following information when you are ready to connect:

Hadoop JobTracker (RHEL)

- Platform: RedHat Enterprise Linux 5 (rhel5 im
- Remote Computer: 152.1.14.186
- User ID: rsridha
- Password: (use your campus password)

Hadoop (RHEL)

- Platform: RedHat Enterprise Linux 5 (rhel5 im
- Remote Computer: 152.1.14.208
- User ID: rsridha
- Password: (use your campus password)

Hadoop (RHEL)

- Platform: RedHat Enterprise Linux 5 (rhel5 im
- Remote Computer: 152.1.14.210
- User ID: rsridha
- Password: (use your campus password)

Figure 22: Screen shot showing the Master and the Slave connection details

Step 3: The master and the slave machines can be accessed using any *ssh client*.

Step 4: On the master machine, download the Pig Latin code base which can be obtained from <http://hadoop.apache.org/pig/>

Step 5: Configuration changes:

- I. On the master system, find the location of the **Conf** folder, and modify the file named “**masters**” to include the IP address of the master system.
- II. Similarly, find the location of the **Conf** folder on every slave machine and modify the “**slaves**” file to include the IP address of the corresponding slave machine.
- III. In all the machines, replace the configuration from the **localhost** to the **machine’s IP address** in the file named “**conf/Hadoop-site.xml**” within the **conf** folder.

```
<property>
    <name>fs.default.name</name>
    <value>hdfs://localhost:9000</value>
</property>
```

IV. The firewall configuration has to be disabled in order to be able to perform the communication between the master and the slave machines.

Thus to bring down the firewall, make the following changes :

In the master machine:

```
chmod 755 bin/*.sh
```

```
sudo bash
```

```
/sbin/service iptables stop
```

```
Exit
```

In every slave machine:

```
chmod 755 bin/*.sh
```

```
sudo bash
```

```
/sbin/service iptables stop
```

```
Exit
```

3. Query execution

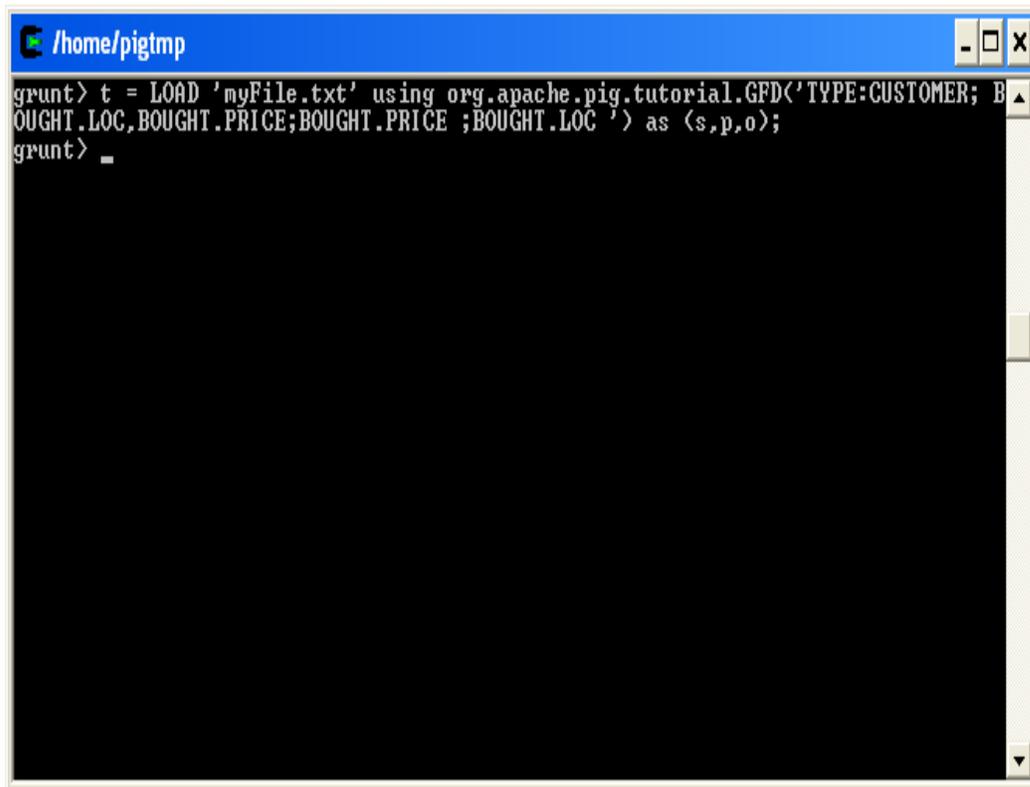
When we execute:

```
$ cd bin/hadoop namenode -format
```

```
$ bin/start-all.sh
```

Simple Pig Latin command line interpreter called the **GRUNT** start its execution.

Using this interface the users can interact to submit the jobs

A screenshot of a terminal window titled "/home/pigtmp". The terminal shows a GRUNT prompt followed by a Pig Latin command: "t = LOAD 'myFile.txt' using org.apache.pig.tutorial.GFD('TYPE:CUSTOMER; BOUGHT.LOC,BOUGHT.PRICE;BOUGHT.PRICE ;BOUGHT.LOC ') as (s,p,o);". The prompt then returns to "grunt> _".

```
/home/pigtmp  
grunt> t = LOAD 'myFile.txt' using org.apache.pig.tutorial.GFD('TYPE:CUSTOMER; BOUGHT.LOC,BOUGHT.PRICE;BOUGHT.PRICE ;BOUGHT.LOC ') as (s,p,o);  
grunt> _
```

Figure 23: Screen shot of the GRUNT interpreter

In the GRUNT command line interface, the user command can be executed.

Few examples are shown below:

- Configure the input folder:
grunt> bin/hadoop fs -put conf input
- Execution of a word count example:
grunt> bin/hadoop jar hadoop-0.17.2.1-examples.jar wordcount

input output
- Execution of a command to LOAD data from the *input.rdf* file:
grunt> inputdata = LOAD 'input.rdf' as (sub,prop,obj);
- Run GFD command:
grunt> inputdata = LOAD 'input.rdf' using

GFD('TYPE:CUSTOMER;

BOUGHT.LOCATION,BOUGHT.PRICE;

BOUGHT.PRICE; BOUGHT.LOCATION) as

(sub,prop,obj);
- Execute GBD command:
grunt> basedata = LOAD 'input.rdf' using

```
GBD('TYPE:CUSTOMER;  
BOUGHT.LOCATION,BOUGHT.PRICE;  
;NULL)as  
(sub,prop,obj);
```

- Execute MDJ command:
grunt> outputdata = LOAD 'MDJ.rdf' using
MDJ('TYPE; SUM:PRICE; LOCATION:NC);
- Command to access the output generated:
grunt> bin/hadoop fs -get output output
grunt> cat output/*
- Exit from the GRUNT command interface
grunt> quit

4. Sample data

Figure 24 shows the screen shot of the sample input data where the subject and the predicate are represented using URIs and the object either is an URI or a literal.

```
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType1> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www4.wiviss.fu-ber
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType1> <http://www.w3.org/2000/01/rdf-schema#label> "Thing" .
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType1> <http://www.w3.org/2000/01/rdf-schema#comment> "The Product Type of all Prod
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType1> <http://purl.org/dc/elements/1.1/publisher> <http://www4.wiviss.fu-berlin.de
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType1> <http://purl.org/dc/elements/1.1/date> "2000-07-04" <http://www.w3.org/2001
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType2> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www4.wiviss.fu-ber
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType2> <http://www.w3.org/2000/01/rdf-schema#label> "amour dupable" .
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType2> <http://www.w3.org/2000/01/rdf-schema#comment> "lengthened galling outposts
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType2> <http://www.w3.org/2000/01/rdf-schema#subClassOf> <http://www4.wiviss.fu-ber
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType2> <http://purl.org/dc/elements/1.1/publisher> <http://www4.wiviss.fu-berlin.de
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType2> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www4.wiviss.fu-ber
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType2> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www4.wiviss.fu-ber
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType2> <http://www.w3.org/2000/01/rdf-schema#label> "amour dupable" .
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType2> <http://www.w3.org/2000/01/rdf-schema#comment> "lengthened galling outposts
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType2> <http://www.w3.org/2000/01/rdf-schema#subClassOf> <http://www4.wiviss.fu-ber
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType2> <http://purl.org/dc/elements/1.1/publisher> <http://www4.wiviss.fu-berlin.de
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType2> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www4.wiviss.fu-ber
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType2> <http://www.w3.org/2000/01/rdf-schema#label> "Thing" .
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType2> <http://www.w3.org/2000/01/rdf-schema#comment> "The Product Type of all Prod
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType2> <http://purl.org/dc/elements/1.1/publisher> <http://www4.wiviss.fu-berlin.de
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType2> <http://purl.org/dc/elements/1.1/date> "2000-07-04" <http://www.w3.org/2001
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType2> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www4.wiviss.fu-ber
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType2> <http://www.w3.org/2000/01/rdf-schema#label> "amour dupable" .
<http://www4.wiviss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType2> <http://www.w3.org/2000/01/rdf-schema#comment> "lengthened galling outposts
```

Figure 24: Screen shot of the sample input data file

Figure 25 shows the screen shot of the sample base data that is generated after the execution of the GBD operator where the subject is represented using an URIs, the predicate has value “**BASE**” and the object is initialized to “**NULL**”.

```

instances/ProductType1_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution5, BASE, NULL)
instances/ProductType1_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution4, BASE, NULL)
instances/ProductType1_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution3, BASE, NULL)
instances/ProductType1_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution2, BASE, NULL)
instances/ProductType1_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution1, BASE, NULL)
instances/ProductType2_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution5, BASE, NULL)
instances/ProductType2_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution4, BASE, NULL)
instances/ProductType2_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution3, BASE, NULL)
instances/ProductType2_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution2, BASE, NULL)
instances/ProductType2_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution1, BASE, NULL)
instances/ProductType3_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution5, BASE, NULL)
instances/ProductType3_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution4, BASE, NULL)
instances/ProductType3_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution3, BASE, NULL)
instances/ProductType3_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution2, BASE, NULL)
instances/ProductType3_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution1, BASE, NULL)
instances/ProductType4_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution5, BASE, NULL)
instances/ProductType4_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution4, BASE, NULL)
instances/ProductType4_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution3, BASE, NULL)
instances/ProductType4_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution2, BASE, NULL)
instances/ProductType4_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution1, BASE, NULL)
instances/ProductType5_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution5, BASE, NULL)
instances/ProductType5_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution4, BASE, NULL)
instances/ProductType5_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution3, BASE, NULL)
instances/ProductType5_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution2, BASE, NULL)
instances/ProductType5_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution1, BASE, NULL)
instances/ProductType6_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution5, BASE, NULL)
instances/ProductType6_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution4, BASE, NULL)
instances/ProductType6_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution3, BASE, NULL)
instances/ProductType6_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution2, BASE, NULL)
instances/ProductType6_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution1, BASE, NULL)
instances/ProductType7_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution5, BASE, NULL)
instances/ProductType7_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution4, BASE, NULL)
instances/ProductType7_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution3, BASE, NULL)
instances/ProductType7_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution2, BASE, NULL)
instances/ProductType7_http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/StandardizationInstitution1, BASE, NULL)

```

Figure 25: Screen shot of the sample base data file

Figure 26 shows the screen shot of the sample output data that is generated after the execution of the MDJ operator where the subject and predicate are represented using an URIs and the Object has the aggregation result.

