ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 3, Issue 2, pp: (1031-1038), Month: April - June 2015, Available at: www.researchpublish.com

Implementation of query System on hadoop using Map reduce and RDF data on Web Semantics

[1]Manoj kumar, [2]Dr. Yashpal singh

[1]M.Tech student, Department of Computer Science & Engg, GITAM, kablana, jhajjar, Haryana
[2]Associate Prof., Department of Computer Science & Engg, GITAM, kablana, jhajjar, Haryana

Abstract: **The Semantic Web is an emerging technology which aims at making data across the globe semantically connected. The data is represented in a very simple statement like construct having a subject, predicate and an object. This can be visualized as a graph with the subject and the object as nodes and the predicate as an edge connecting the two nodes. When many statements like these are collected together they forms an RDF graph. There are RDF query languages to query such data, and SPARQL is one of them. According to the SP^2 Bench performance benchmarks, the SPARQL queries are very slow for RDF data with millions of triples. Hence, we aim to develop a Implementation of query System on hadoop using Map reduce technique and RDF Data model of parallelization and hypothesize that this system will outperform the scalability and performance reported by the SP^2 Bench. We extend ARQ, an open source SPARQL query engine provided by the Jena framework, to work with the Hadoop Map Reduce framework and implement distributed SPARQL query processing. This thesis provides the detailed implementation and algorithmic details of our work. We contribute two novel methods to optimize RDF query engine which exploits document indexes and a join pre-processing technique. The experimental results show the merits and demerits of using Map Reduce for distributed RDF query processing and provides us a clear path for future work.**

Keywords: **Hadoop framework, Cassandra key-value, RDF Dataset, MapReduce, SPARQL, Jena framework, Turtle, RDFa, Triplestore, Quadstore. HDFS, N-Triples, SP^2 Bench, Thrift, Hector.**

I. INTRODUCTION

The world wide web, in its current form, presents data in a way that is understandable and consumable by human users. The same level of semantic understanding is not yet a capability of machines. A computer program can read a web page and understand its formatting and layout, but cannot easily extract greater meaning from its content. As the web grows to encompass large portions of human knowledge, and vast amounts of data, the individual user is limited in their search for information simply by the overwhelming amount accessible. Machines can help today through the use of search engines, but even the most advanced engines available must use sophisticated data mining and interpretation processes to make even a modest attempt at lifting semantic meaning from data that does not explicitly express this. The goal of the semantic web is to alleviate this issue by more directly including machine processable semantics into the data of the web. With this data present, machines can assist human queries by inferring information for them from stores of data too large to search and process manually.

The Resource Description Framework (RDF) [3] is a family of standards and specifications developed by the World Wide Web Consortium (W3C) to support the notion of a semantic web. Through these specifications RDF details a way to make statements about resources such that a machine could understand them. The primary mechanism for doing so in the RDF data model is through the use of a triple, (S; P; O), consisting of a subject, a predicate, and an object. In this way statements can be made about the properties of a subject, and the values of those properties. For example one could express that Aristotle (the subject) was influenced by (the predicate) Plato (the object). These triples form a labeled directed multi graph with a single subject having potentially many predicates, and any given object also acting potentially as the subject of other statements.

The mere presence of appropriate RDF data does not, however, enable the semantic web. This data must be able to be queried in such a way as to allow for complex searches and inferences to be made. SPARQL is a declarative query language for RDF that bears some syntactic resemblance to SQL. A SPARQL query in its most simple form consists of a SELECT and a WHERE clause. The WHERE clause contains what is referred to as a Basic Graph Pattern (BGP). A BGP is a set of triple patterns which are very similar to the data triples described above, with the addition of the allowance for variables. In SPARQL a variable is a name proceeded by a question mark, as in ?varName. An example SPARQL query for finding all people that Aristotle was influenced by and in turn influenced him, appears in figure 1

```
SELECT? Influncer
Where
{
<http://dbpedia.org/resource/Aristotle> <http://dbpedia.org/ontology/influencedBy> ?influencer.

?influencer<http://dbpedia.org/ontology/influencedBy><http://dbpedia.org/resource/Aristotle>
}
```

Figure 1: A simple SPARQL query for finding both the people that Aristotle influenced and those who he was influenced by

The text denoting “Aristotle" and “influencedBy" are IRI's, which are generalizations of Uniform Resource Identifiers (URI) A URI is in turn a generalization of the familiar URL, with the loosening of the restriction requiring the identifier to actually represent a locatable web resource. The verbose nature of the statement above is in practice mitigated via the use of prefixes to represent the base of a URI. http://dbpedia.org/resource/Aristotle could, for example, become dbpr:Aristotle through the use of a prefix binding “dbpr" to \http://dbpedia.org/resource/\. In general the BGPs can contain 1 to many rows with 0 to many variables.

Processing a BGP over a set of RDF data can take many forms. It is not uncommon to model the RDF data in a relational schema and process a query by either writing the query directly in SQL, or by transforming a SPARQL query to relational algebra as in the method described by [4]. Dedicated triplestore databases also exist for storing and querying RDF triples. Various hurdles exist with these and other approaches when processing large datasets. In the relational model RDF data must be transformed into tabular form and queries must be either specifically written for the particular schema used (removing portability), or must be translated from SPARQL to SQL before execution (to a lesser extent still removing portability as the translation engine must be aware of the relational schema in use). Triplestores present an intriguing option as no paradigm shift is needed between the native representation of the data, and its logical storage. These tools, however, are still relative new comers to the landscape and as such the features, scalability, and selection are not where their relational counterparts are today. In both cases (as well as a handful of other mechanisms such as key-value stores) the data must first be loaded into the database before processing can begin. In many situations a given data set must be queried, but there is no need to persist the data after the results have been returned. Even in some of the more advanced triple stores, loading of data sizing in the billions of triples is measured in hours [6].

Map-Reduce is a data processing model developed at Google for efficiently handling very large datasets using a fault tolerant mechanism. It is inspired by principles of functional programming, though the common map and reduce functions take on slightly new meanings when interpreted in the Map-Reduce paradigm. Map-Reduce allows for an input file to be split into many independent subprocessing steps, each of which could be handled by a separate machine (or processor, or core). The results of these sub-processes are then reduced back down to a single, or a set of, solutions. This framework allows for massive scalability, along with a simple progamming model that does not ask developers to think explicitly about parallelism or concurrency issues. Map-Reduce has a proven track record of efficiently handling large data queries and is quickly moving into main stream use within both academia and industry. The concepts be-hind Map-Reduce have moved outside of Google, and take form in various implementations, the most well known being the open source framework Hadoop [1].

BGP processing via Map-Reduce holds great promise. The intrinsic scalability of Map-Reduce make it ideal for the potentially very large, web-scale, RDF datasets. Map-Reduce, in the general case, unfortunately does not handle join style operations naturally in an efficient way. Through the use of a platform called Pig [2], and its associated Pig Latin language, joins have been made simpler for the developer to describe, but the processing inefficiencies remain unchanged.

Other work in this area, such as [9], has described algorithms for using Map-Reduce in an iterative fashion to process SPARQL queries. This defines a process for an initial Map-Reduce pass to perform selections from the data based on the query, and then iteratively apply a different Map-Reduce job to effectively execute join operations. [9] introduces ideas for reducing the number of joins required, and thus the number of iterations, but in most cases multiple joins will still be required. Map-Reduce iterations can be quite heavy in terms of overhead. To optimize execution time, ideally a single pass through Map-Reduce would be implemented. In [11] the authors introduce a new phase to Map-Reduce that they refer to as Merge. The new Map-Reduce-Merge paradigm is intended to allow for traditional Map-Reduce to be extended in such a way as to more adequately handle join style operations. While no implementation was publicly available, the high level concepts from Map-Reduce-Merge can be applied to SPARQL query processing. This paper details algorithms for handling SPARQL queries via a method similar to Map-Reduce-Merge, and presents a simple implementation of these algorithms.

II. ALGORITHM FOR BGP PROCESSING

By applying a 4-phase algorithm to a given SPARQL query and RDF triple dataset, a single pass Map-Reduce execution can be achieved. This relieves the overhead of iterative Map-Reduce executions. Such overhead includes, but is not restricted to the need to write each potentially large intermediate data set to disk, and then read it back in. A query containing n variables (after any reductions have occurred) could result in up to n executions of Map-Reduce when run in an iterative Map-Reduce environment. Where Map-Reduce is primarily applied to problems involving large datasets, even the intermediate results could be measured in gigabytes thus reading and writing these data multiples times contributes in a non-trivial way to the overall runtime.

The 4-phase algorithm consists of:

1. Preprocessing Phase
2. Map Phase
3. Reduce Phase
4. Merge Phase

Preprocessing Phase: The preprocessing phase receives as input a SPARQL query, and reference to the location of the dataset to process. This phase involves completing 3 main tasks:

1. Translate the SPARQL query to SPARQL algebra
2. Compute a "minimal cover”
3. Identify the "overlap rows”

Task 1 involves translating the SPARQL query in text form into a representation of SPARQL algebra (here in tree form). SPARQL algebra is similar to relational algebra in intent, and partially in form. In the implementation created as part of this work, only two of the operators from SPARQL algebra are considered, BGP , and projection. The remaining operators remain as future work, but represent less critical components of the processing from a complexity perspective.

Once the BGP's have been identified, they can be processed to determine results for the 2nd and 3rd tasks of the preprocessing stage. First what this work refers to as a minimal cover is computed. Conceptually a minimal cover is a subset of variables from the query that must be considered as drivers for the subsequent phases of processing. All phases of this algorithm have runtimes which are in part a function of the number of variables. By reducing the number of variables to consider, the process can be made more efficient by a substantial factor. Removing a single variable from an n variable solution will cause the later 3 phases to work with a value of n 1 rather than n in M intermediate results. Here M is the number of triples in the input, and is a selectivity percentage of the variables over the input.

To perform this reduction we identify a minimal cover, being the minimal set of variables that will cover the BGP. Here minimal cover is defined with respect to a BGP which is interpreted as a graph whose nodes are the elements of each triple in the triple patterns in the BGP, and are arranged in rows corresponding to each triple pattern. A minimal cover is then defined as:

- A path through the BGP such that:
  - An edge can exist between two nodes on the same row if both nodes represent a variable (of the same or different name).
  - An edge can exist between two nodes on different rows only if the corresponding nodes are both variables with the same name.
- The path is a cover, meaning that it has at least one node on each row of the BGP.
- Removing any element from the BGP causes it not to be a cover.
- No shorter path holding the same properties as above exists.

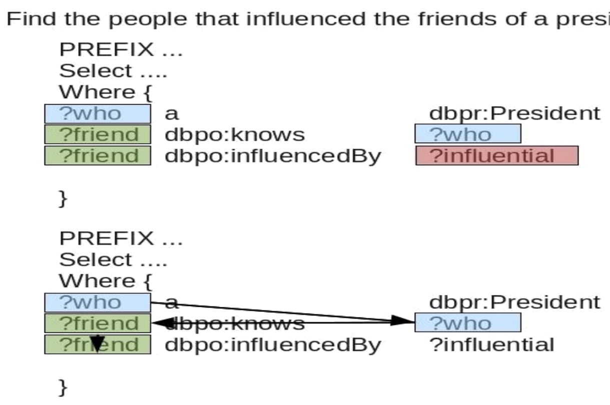
Various methods exist for finding such a minimal cover. The method selected is of little consequence to the overall algorithm as this step is performed only once, and typical queries will have fewer than 20 variables.

Figure 2 depicts an example minimal cover. Here a cover can move vertically from variable ?who on the first line of the BGP to ?who on the 2nd line. It can then move horizontally to ?friend followed by another vertical jump to ?friend on the 3rd line and a horizontal jump to ?influential. This is indeed a cover, but not a minimal cover as ?influential is not required by the definition. ?influential would thus be identified for removal as the set f?who; ?friends is sufficient as a cover for this BGP.

A minimal cover may include the entire set of variables in the BGP in the case where all variables must be included to form a cover. Further, in some extreme cases a cover may not exist at all. In this case the result is essentially the Cartesian product of each variables' matches, and no reduction is possible.

Those variables removed from the set under consideration must still be considered, but during the processing of those included in the minimal cover, and not in addition to. These removed variables do not contribute to the size of the intermediate results, and thus their impact on the overall runtime is minimal. See below, in Reduce Phase, for a description of how these variables are used.

In the 3rd task of the preprocessing stage, overlap rows are identified. These are simply the rows where two or more variables are present on a single row in the triple patterns forming the BGP. This information is used later in the process when performing the join between single variable solutions.



```
PREFIX ...
Select ....
Where {
?who      a                     dbpr:President
?friend   dbpo:knows            ?who
?friend   dbpo:influencedBy     ?influential

}
PREFIX ...
Select ....
Where {
?who      a                     dbpr:President
?friend   dbpo:knows            ?who
?friend   dbpo:influencedBy     ?influential

}
```

Figure 2: An example minimal cover, where ?influential is removed

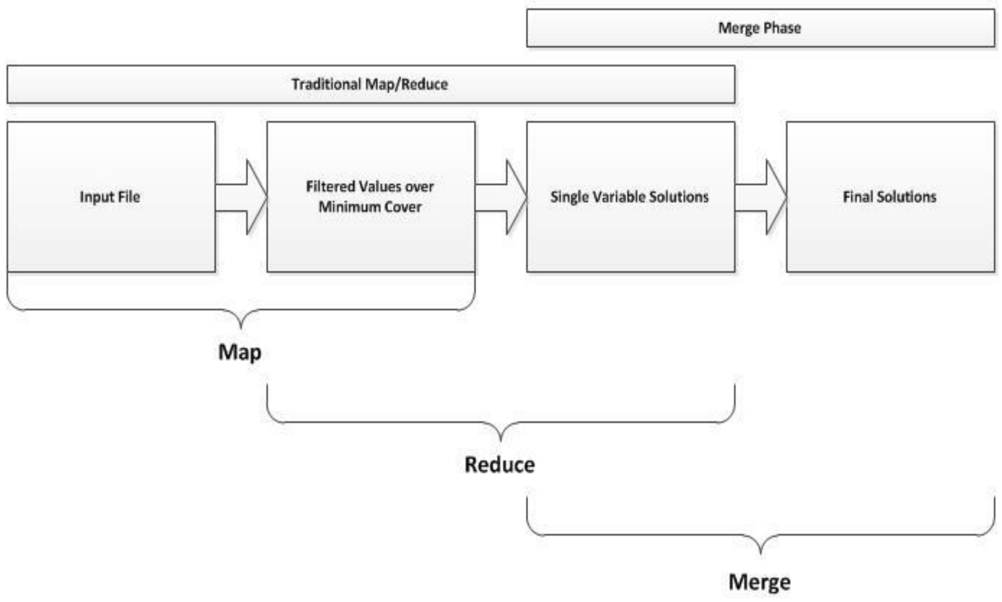
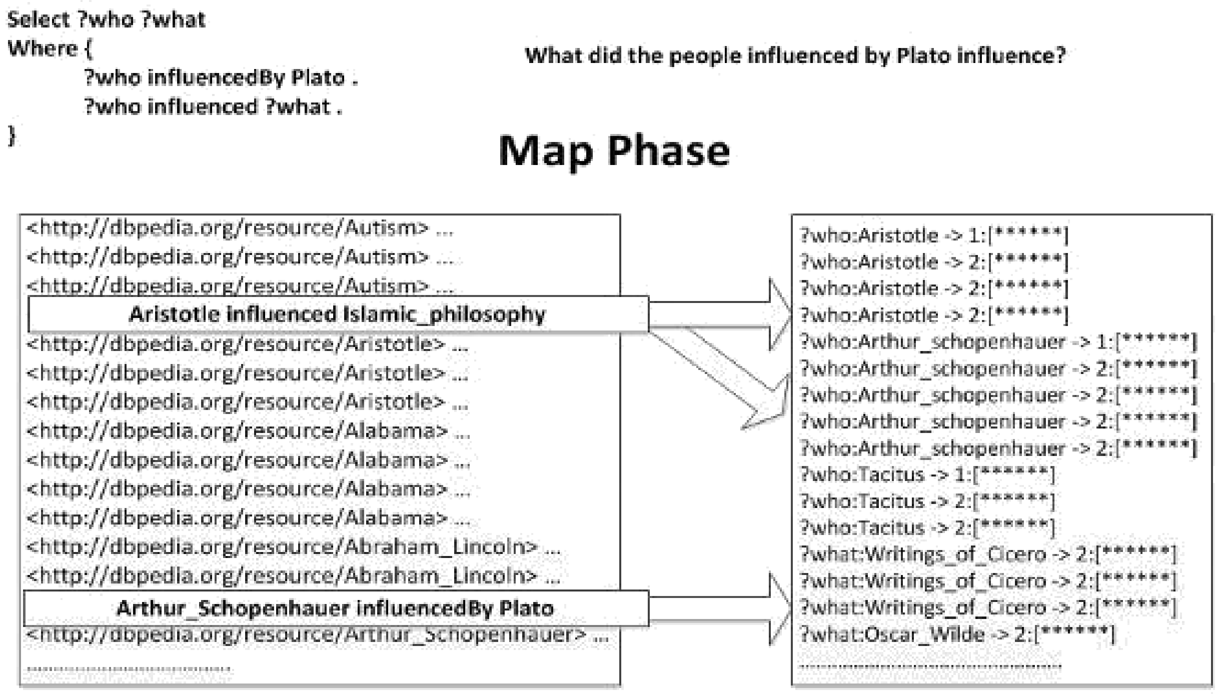


Figure 3: An overview of the last 3 phases


What did the people influenced by Plato influence?
Map Phase
Aristotle influenced Islamic_philosophy
Arthur_Schopenhauer influencedBy Plato

```
Select ?who ?what
Where {
        ?who influencedBy Plato .
        ?who influenced ?what .
}
```

```
<http://dbpedia.org/resource/Autism> ...
<http://dbpedia.org/resource/Autism> ...
<http://dbpedia.org/resource/Autism> ...
<http://dbpedia.org/resource/Aristotle> ...
<http://dbpedia.org/resource/Aristotle> ...
<http://dbpedia.org/resource/Aristotle> ...
<http://dbpedia.org/resource/Alabama> ...
<http://dbpedia.org/resource/Alabama> ...
<http://dbpedia.org/resource/Alabama> ...
<http://dbpedia.org/resource/Alabama> ...
<http://dbpedia.org/resource/Abraham_Lincoln> ...
<http://dbpedia.org/resource/Abraham_Lincoln> ...
<http://dbpedia.org/resource/Arthur_Schopenhauer> ...
.................................
```

```
?who:Aristotle -> 1:[******]
?who:Aristotle -> 2:[******]
?who:Aristotle -> 2:[******]
?who:Aristotle -> 2:[******]
?who:Arthur_schopenhauer -> 1:[******]
?who:Arthur_schopenhauer -> 2:[******]
?who:Arthur_schopenhauer -> 2:[******]
?who:Arthur_schopenhauer -> 2:[******]
?who:Arthur_schopenhauer -> 2:[******]
?who:Tacitus -> 1:[******]
?who:Tacitus -> 2:[******]
?who:Tacitus -> 2:[******]
?what:Writings_of_Cicero -> 2:[******]
?what:Writings_of_Cicero -> 2:[******]
?what:Writings_of_Cicero -> 2:[******]
?what:Oscar_Wilde -> 2:[******]
..........................................
```

Figure 4: An example snapshot of the map phase

Map Phase: Figure 3 depicts a high level view of the last 3 phases of the algorithm. The map phase consists of the left hand two boxes. Here the input data is filtered over the minimal cover defined above and output rows are sent to the next phase of processing. Map proceeds by checking each row of the input le and comparing it against the rows of the BGP from the query. A match exists for a data row and a BGP row if the subjects, predicates, and objects match, or where not matching is a variable. Figure 5 depicts this.

A given data row may match 0, 1, or multiple BGP pattern rows, and thus 0, 1, or multiple output rows may be generated for any given input row. When a match is found a key value pair is output in the form:

Key: (variableName; matchedValue)

Value: (triplePatternIndex; dataRow)

Here triplePatternIndex corresponds to the index in the BGP of the triple pattern that matched (i.e. the 3rd row down in the where clause will have index 3, or 2 in a 0-based index). These key value pairs progress on to the reduce phase, and all other data is disregarded.

Figure 4 depicts a snapshot of an example match phase. Here BGP pattern 1: {?who influencedBy Plato}, has matched [Arthur_Schopenhauer influencedBy Plato]. ?who was bound to Arthur_Schopenhauer, and the other two elements matched exactly. Similarly [Aristotle influenced Islamic_Philosophy] has matched the 2nd row of the BGP, {?who influenced ?what}. ?who was bound to Aristotle, and ?what was bound to Islamic_Philosophy.

Reduce Phase: The reduce phase of the algorithm receives the output from the map phase, grouped such that for any given key an instance of the reduce operation receives that key along with all values associated with it. During the map phase a particular variable may have been bound to the same value for multiple different data rows. Using the query from Figure 4 the variable ?who may have been bound to Aristotle for the data row [Aristotle influenced Islamic_Philosophy] as well as the data row [Aristotle influencedBy Plato]. An instance of the reduce operartion would then receive the key (?Who, Aristotle) and the list of values ((0,[Aristotle influencedBy Plato]),(1,[Aristotle influenced Islamic_Philosophy])), among possibly others.

The input values will be partitioned based on the BGP row index (0, and 1 in the example above), and the possible combinations of values from each partition will be checked for single variable completeness. For a set of data rows to have the property of single variable completeness the rows will have matched all triple patterns in the BGP containing a particular variable, with all bindings for that variable having the same value. In the above example the variable ?who appears on two rows in the BGP. Since the first occurance is matched by

[Aristotle influencedBy Plato] and the second by [Aristotle influenced Islamic_Philosophy], with both binding ?who to Aristotle, these triples form a single variable solution. Single variable solutions are output to the next phase of the algorithm.

A further qualification for a single variable solution is that any other variables occurring more than once within the BGP rows under consideration must have the same value in all locations. For example, a variable ?x may appear on 4 rows of a BGP in a query. If somewhere among those 4 rows the variable ?y occurs twice, any single variable solution for ?x must have a consistent value for the occurrancecs of ?y. These variables include both those in the minimal-cover, and those that were removed.

Φ subject = ψsubject V isVariable (ψsubject)
Φpredicate = ψpredicate V isVariable (ψpredicate)
Φobject = ψobject V isVariable(ψobject)

Figure 5:- Matching in map phase

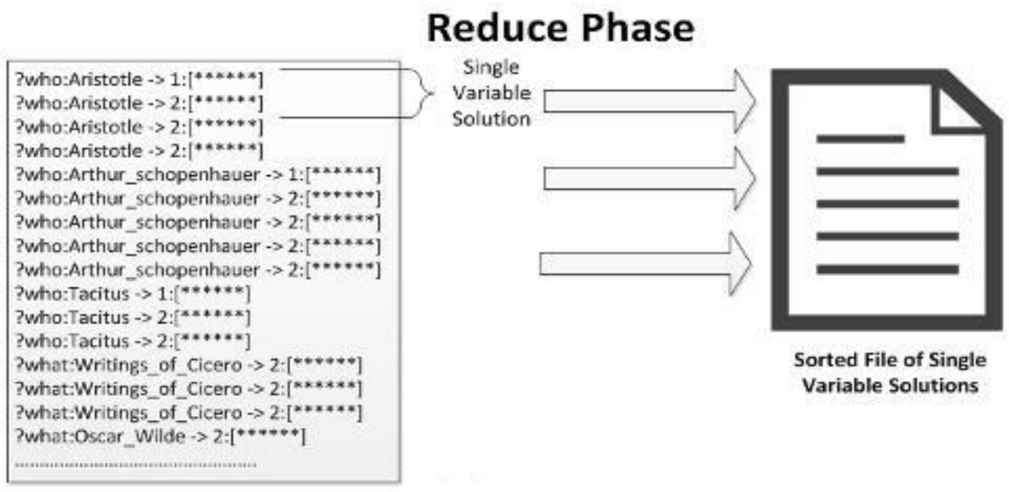

Reduce Phase
?who:Aristotle -> 1:[******]
?who:Aristotle -> 2:[******]
?who:Aristotle -> 2:[******]
?who:Aristotle -> 2:[******]
?who:Arthur_schopenhauer -> 1:[******]
?who:Arthur_schopenhauer -> 2:[******]
?who:Arthur_schopenhauer -> 2:[******]
?who:Arthur_schopenhauer -> 2:[******]
?who:Arthur_schopenhauer -> 2:[******]
?who:Tacitus -> 1:[******]
?who:Tacitus -> 2:[******]
?who:Tacitus -> 2:[******]
?what:Writings_of_Cicero -> 2:[******]
?what:Writings_of_Cicero -> 2:[******]
?what:Writings_of_Cicero -> 2:[******]
?what:Oscar_Wilde -> 2:[******]
Single Variable Solution
Sorted File of Single Variable Solutions

Figure 6: An example snapshot of the reduce phase

Figure 6 provides a snapshot of the reduce phase. Here we see that since ?who bound to Aristotle matched patterns 1 and 2, and those are the only patterns in the BGP referencing ?who, that a single variable solution is output. Note this gure abbreviates the data row values as ***** to conserve space.

At the end of the reduce phase the typical Map-Reduce job is done, and output is written to the file system. A true Map-Reduce-Merge system may remove the requirement for a full write to disk at this stage, and could potentially even include a type of pipelining to allow the Merge phase to begin before Reduce has fully completed. In the implementation for this work, such an advanced system was not available, and the creation of one while related, was outside the immediate scope. Thus the implementation, to be described in section III, adds the Merge phase as a separate entity to be run after the completion of normal map-reduce operations complete.

Merge Phase. At the end of the Reduce phase a sorted list of single variable solutions are output. Each single variable solution represents a set of data rows that would satisfy a single variable within the BGP. At least one single variable solution must exist for each variable in order to produce a non-empty result set. An optimization can thus be made that keeps a flag for each variable seen during the map phase, and if not all variables had a result when beginning the reduce phase, processing can stop and an empty result set returned.

Assuming that each variable does have at least one single variable solution, the Merge phase proceeds During the Preprocessing phase overlap rows were determined. These are just the rows where more than one variable from the selected minimal cover is present. These are used here in the Merge phase as the determining factor when attempting to join any two single variable solutions.

A join now occurs between the sets of single variable solutions (with each set consisting of those single variable solutions for the same variable). This phase can borrow significantly from the RDBMS domain in the actual implementation of the join algorithm. For example the reduce phase outputs the results in sorted order, and can do so based on the largest expected join condition, meaning the majority of the joins to occur can use a sort-merge join on pre-sorted datasets. Given a low selectivity rate, the sets being joined will be small at this point as filtering in previous stages has pruned the input to only those rows that are at least partial matches for the result.

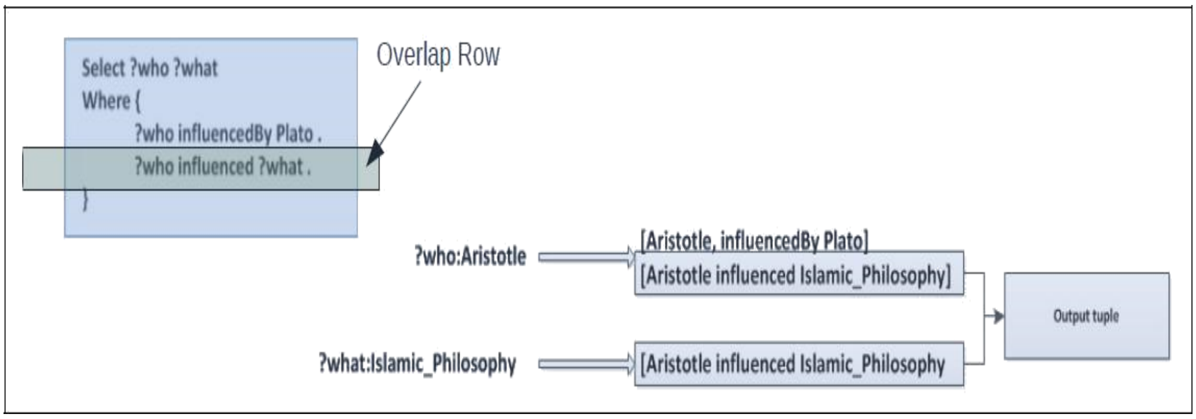

Select ?who ?what
Where {
?who influencedBy Plato .
?who influenced ?what .
}
Overlap Row
?who:Aristotle
[Aristotle, influencedBy Plato]
[Aristotle influenced Islamic_Philosophy]
?what:Islamic_Philosophy
[Aristotle influenced Islamic_Philosophy
Output tuple

Figure 7: An example snapshot of the merge phase

In Figure 7 because the two highlighted rows match the only overlap row in the query, they move to the output. Output rows from this phase will be exactly those that should be represented in the final result. A final step while output is generated is to simply project these results over any projection criteria from the SELECT clause of the query.

III. IMPLEMENTATION

An implementation, MRSPARQL, was developed to demonstrate and test these algorithms. It provides a simple web interface allowing the user to select a dataset to work with, and input a SPARQL query. This version is restricted to working with queries that result in projections over basic graph patterns (BGP). The result set will be returned to the user and displayed in tabular form as HTML.

Technologies: MRSPARQL is written in Java 1.6, using simple Java EE servlets and Java Server Pages (JSP) for the web interface. The core Map-Reduce functionality makes use of Hadoop 2.3[1]. The parsing engine makes use of ARQ [7], a SPARQL processing library for Jena [8], in extracting SPARQL algebra from textual Data. The data used while building and testing MRSPARQL comes from DBPedia [5], an extract of data in RDF formats from Wikipedia.

Experimental Results: To date MRSPARQL has focused on correctness, as opposed to efficiency. While the algorithm above was designed for efficiency, it's implementation in MRSPARQL made use of the most intuitive sub-algorithms. There is great room for improvement here, as is mentioned in section IV. In addition MRSPARQL has been tested on a single core, singe processor Intel Core i3 2.66GHz. This fact completely restricts Hadoop's ability to implement the map and reduce operations in a parallel or distributed manner the core of Hadoop's efficiency. Due to these facts experimental results have only been used to test for correctness, and to prove the viability of map-reduce-merge as a platform for processing SPARQL queries. Here initial test results will be brie y mentioned, but any comparisons to other systems are withheld pending testing on more realistic distributed systems, and using a more optimized implementation. Using an extract of the DBPedia "Ontology Infobox Types" dataset, which totals approximately 845 MB, and contains 6,173,940 triples, a moderately complex SPARQL query resulted in a runtime of 35 seconds. A similar query run against the "Ontology Infobox Properties" DBPedia extract which totals 1.7 GB and 13,795,664 triples had an average runtime of 79 seconds. Here again we emphasize the results are from an un-optimized , single processor execution and are presented with the intent of showing the promise these algorithms hold in more appropriate environments.

IV. FUTURE WORK

Future work can be broken down into three parts Algorithms, Implementation and Testing.

Algorithm: The algorithms describes here can be extended to encompass the complete set of SPARQL algebra operators , allowing for a more robust system. BGP forms the core of any SPARQL query, but extending to the full operator set would be required for any real use system.

Implementation: The implementation in its current form uses sub-optimal solutions to many tasks it performs while executing the above algorithms. Updating this to provide more efficient execution can likely decrease the runtime many fold.

Testing: Given a more robust implementation, complete benchmark testing would be required. A benchmark system such as LUBM [10] would provide adequate data, and enable useful comparisons among other frameworks. For a realistic test this system should be benchmarked in the cloud or another suitable distributed environment where Hadoop can benefit from concurrent processing.

V. CONCLUSIONS

By removing the need for iterative executions of Map-Reduce we can approach what appear to be similar run-times to current systems even in a single processor, non-optimized environment. This fact points to promising results when applied in an appropriate setting. The steps outlined in section IV will be performed in order to make conclusive statements.

REFERENCES

[1] Apache. Apache hadoop. http://hadoop.apache.org/, 2011.

[2] Apache. Apache pig. http://pig.apache.org/, 2011.

[3] World Wide Web Consortium. W3C rdf. http://www.w3.org/RDF/, 2011.

[4] Richard Cyganiak. A relational algebra for SPARQL. Technical report, Digital Media Systems Laboratory, HP Laboratories Bristol, 2005. Tech. Rep. HPL-2005-170, 2005 - http://www.hpl.hp.com/techreports/2005/HPL-2005-170.pdf.

[5] Wikipedia community DBPedia. DBPedia 3.6 downloads. http://wiki.dbpedia.org/Downloads36, 2011.

[6] Franz Inc. Allegrograph rdf store - high performance storage. http://www.franz.com/agraph/allegrograph/, 2011.

[7] HP Labs. Arq a sparql processor for jena. http://jena.sourceforge.net/ARQ/, 2011.

[8] HP Labs. Jena semantic web framework. http://jena.sourceforge.net/, 2011.

[9] Jaeseok Myung, Jongheum Yeon, and Sang-goo Lee. Sparql Basic Graph Pattern Processing with Iterative MapReduce. In Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud, 2010.

[10] Lehigh University. Lehigh university benchmark. http://swat.cse.lehigh.edu/projects/lubm/, 2011.

[11] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-Reduce-Merge: Simpli ed Relational Data Processing on Large Clusters. In Proceedings of the 2007 ACM SIGMOD international conference on Management of data, 2007.