Using Hadoop To Implement a Semantic Method Of Assessing The Quality Of Research Medical Datasets

Stephen Bonner
University Of Huddersfield
Huddersfield, UK
s.bonner@hud.ac.uk

Prof. Grigoris Antoniou
University Of Huddersfield
Huddersfield, UK
g.antoniou@hud.ac.uk

Dr. Laura Moss
University of Glasgow
Glasgow, UK
Laura.Moss@glasgow.ac.uk

Ibad Kureshi
Durham University
Durham, UK
ibad.kureshi@durham.ac.uk

Dr. David Corsair
University of Aberdeen
Aberdeen, UK
dcorsar@abdn.ac.uk

Illias Tachmazidis
University of Huddersfield
Huddersfield, UK
U1273221@hud.ac.uk

ABSTRACT

In this paper a system for storing and querying medical RDF data using Hadoop is developed. This approach enables us to create an inherently parallel framework that will scale the workload across a cluster. Unlike existing solutions, our framework uses highly optimised joining strategies to enable the completion of eight separate SPAQL queries, comprised of over eighty distinct joins, in only two Map/Reduce iterations. Results are presented comparing an optimised version of our solution against Jena TDB, demonstrating the superior performance of our system and its viability for assessing the quality of medical data.

Keywords

RDF, SPARQL, Hadoop, Map/Reduce, Medical Data, Error Checking

1. INTRODUCTION

Recent technological advances in modern healthcare have led to a vast wealth of patient data being collected. This data is not only utilised for diagnosis but also has the potential to be used for medical research. However, according to [5], there are often many errors in datasets used for medical research. In this study they found error rates ranging from 2.3% to 26.9% in a selection of medical research databases. With such high error rates present in medical databases there is clear need for a system to assess the quality of data before it used as the basis of cutting edge research.

1.1 Big Data and Healthcare

Big Data is a rapidly increasing area of interest within the computer science field and draws upon areas from many different disciplines. Both academic and industrial fields are generating and collecting data at an unprecedented rate and scale. This data, and the analysis of it, is being used to replace models and guesswork as the basis for decision making [1]. The biomedical and general healthcare fields have the potential to be one of the biggest contributors to and benefactors from the big data phenomenon. The volume of worldwide healthcare data as of 2012 is estimated to be over 500 petabytes [4]. Analysing this data has the potential to alter biomedical research and healthcare diagnosis [6].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM http://dx.doi.org/10.1145/2640087.2644163 ...$15.00
http://dx.doi.org/10.1145/2640087.2644163.

1.2 Current Implementation

A Linked Data approach to assessing the quality of medical data has been created. [3] This framework can be broken down into three key stages: firstly pre-existing medical data is converted into RDF data and stored in Jena TDB. Secondly the data can be annotated with provenance information, such as the specification of the machines which recorded the data. Lastly a data checking component assesses the quality of the data via a series of eight SPARQL rules. The current implementation relies on traditional semantic web technologies which are not well-adapted to process the volume of data which healthcare entails. Due to this the current framework suffers from performance and scalability issues when processing massive RDF datasets.

2. HADOOP IMPLEMENTATION AND ALGORITHM DESIGN

2.1 Structure and Distribution Of The Real-World Medical Data

For any new framework to be designed, real-world medical data was required. Due to ethical and privacy reasons it was not possible to have access to large volumes of real medical data. However, three anonymised datasets were provided by the University Of Glasgow. The datasets all contain neurological data from the BrainIT project [2]. The datasets contain information regarding neurological readings taken from patients, along with information regarding the machines and sensors which produced these values. The provided datasets represent a small period of time for three different patients and combined they contain 7,933,649 RDF triples. To better understand the distribution of the data, the number of unique RDF Subjects, Predicates and Objects were obtained using a simple Map/Reduce job. The results of which can be seen in table 2. The results show that only 61 unique

predicates are used throughout all of the datasets. Further, it can be seen that the number of unique subjects and objects are closely matched. It also shows that compared to the total number of triples (7,933,649), many of the subjects and objects are replicated numerous times.

All of the original SPARQL queries join distinct groups of triples, which are linked by a common theme. Broadly, the distinct groups of triples are related to two things: patient recording values and permitted value ranges. For any possible optimisations for later queries, knowledge of the distribution of these triple groups would be required. In order to explore the distribution of these groups a Map/Reduce algorithm was devised that assessed the number of members in set triple groups. This was then run against the same three medical RDF datasets, grouping the triples into one of three groups. The groups were ?range, ?obs and ?cs and were chosen to reflect the triple groups used in the SPARQL queries. The result is shown in table 3. The results show that the distribution of the triple groups is massively skewed towards the ?obs group.

As the framework needed to be tested against massive datasets, one billion triples were synthetically generated. To produce the new data, a Map/Reduce algorithm was developed which generated new RDF data based upon an input of real world medical RDF data. The algorithm retains the structure and distribution from the real world data, but inserts new randomly generated values for the variable triple elements.

2.2 Query Planning

Before designing the implementation, a plan to complete all the queries was created. Firstly as the queries share many common elements, a super-query was created to avoid the re-computation of joins. Using this super-query removes a large percentage of duplicated joins which the standard queries would perform. The listing in appendix B.1 below shows how all the required SPARQL queries could be compiled into a single list. While this query would not return the correct result if run upon a standard SPARQL endpoint, using Hadoop enables the correct joins to be performed. Figure 1 shows a graphic representation of how the various triple groups are linked and how the Hadoop-based approach process the required joins. All the elements in a triple group are linked via a single common element, with links between groups also being between common elements.

2.3 Hadoop-Based Framework Design

A key consideration when designing the Hadoop based approach was the joining strategies to be used. The majority of the existing approaches rely on a series of cascading reduce-side joins. Due to the vast number of joins required to assess medical data, an approach similar to the existing solutions would result in many Map/Reduce iterations being used. This in turn would result in very poor performance. From reviewing the current literature it is possible to see that there are gaps which would enable a highly optimised system that stores and quires RDF data using Hadoop to be created. Firstly, the case can be made for exploring a system where prior knowledge of the data to be stored informs a highly optimised system. Knowledge of the structure and distribution of any RDF data could be utilised to enable efficient map-side and broadcast joins to be used. For example, triples which contain common elements could be grouped together to enable map-side joins. Also smaller groups of triples which are required to be joined to larger groups could be made available via a broadcast join. This would save on numerous costly iterative reduce-side joins which the current solutions rely upon. Secondly, the case can be made for designing a system in which knowledge of the type of SPARQL queries to be performed allows for optimised query planning. Currently none of the existing system use knowledge of the SPARQL queries which will be performed to save on costly re-computation. This means that the current solutions would recompute all the required joins for two SPARQL queries in which only one join was different. A highly optimised solution would not waste time and resources re-performing joins, instead it would only perform the additionally required join. This could be achieved by creating a super-query from an input of multiple queries, so that common joins would not be recomputed.

3. DATA UPLOAD ALGORITHM

3.1 Stage One - Compression

Due to the costly nature of dictionary encoding, an alternative solution was developed. To implement this new solution, common predetermined namespaces in the original RDF data are located and replaced with shorter ones. This has the advantage of both reducing the amount of space each triple consumes on the HDFS and also reducing the amount of network traffic during the shuffle and sort phase. As the namespaces to be replaced are predetermined this stage can be implemented as a Map only job. This approach was chosen instead of a full dictionary encoding as it can be accomplished in a map only job and is therefore quicker. It also requires no additional Map/Reduce stage to decode the data once complete, as is required by the dictionary encoding method.

3.2 Stage Two - Sort-On Subject

The goal of this stage is to store all of the RDF predicates and objects for a certain subject on the same line of input on the HDFS. As shown in section 2.1, the majority of subjects used in the medical data are used in multiple unique triples. Utilising a method similar to the one described by [8], can be justified both by this replication of subjects, also many of the joins required for the SPARQL queries are performed upon the subject. Using this method reduces replication of triples to further save space on the HDFS and also enables faster performance of joins in the query stage. The sort on subject stage will allow the query stage to perform joins via a map-side join rather then the more costly reduce-side join method. To implement the sort on subject stage, the Map stage scans the entire compressed RDF data set, setting each RDF subject as the key and the rest of the triple as the value. The Reduce stage then outputs the subject followed by all the associated predicates and objects.

4. QUERY STAGE

The query algorithm is designed to replicate the results that would be returned by running the SPARQL queries used in the original study [3] [7]. To achieve this, several different Hadoop joining strategies were utilised along with exploitation of the distribution of the data.

As discussed in section earlier, the distribution of the medical RDF data can be categorised into two sections. Firstly a

portion of the RDF triples contain information about medical equipment, sensor accuracy, permitted data ranges and medical conditions. This information is a very small proportion of the datasets and is consistent across them all. Secondly the bulk of the RDF data concerns the actual time domain records and values associated with the patients themselves. Knowledge of this meant that it was possible to join the smaller selection of triples to larger ones via a broadcast join. This was achieved by using the Hadoop feature called Distributed Cache. The Distributed Cache allows set files from the HDFS to be pushed to either a Map or Reduce task. This allows multiple elements to be joined via a broadcast join, thus bypassing the need for a series of Cascade Reduce-Side Joins.

4.1 Selection Stage

This stage has two main functions and is implemented as a complete Map/Reduce iteration. Firstly it traverses all triples stored on the HDFS and selects only those required by the queries. Secondly, it then performs some of the required joins before the data is passed to the second stage of the query algorithm. The complete workflow for the selection stage can be seen in figure 4. This figure will be referred to throughout this section.

The input to this stage is the formatted data emitted by the upload stage and stored on the HDFS. As the data query algorithm formats the data so that all predicate and object for a certain subject are available on the same line, this enables a map-side join to be used to perform some of the required joins. The requirement for a map-side join is that triples needing to be joined share a common subject. Map-side joins are the most efficient of the available Hadoop joining strategies as they bypass the need for any of the data to be transferred over the network. An example section from a SPARQL query which could be joined via a map-side join is shown in figure 2.

Figure 2 shows the ?range portion of query which is common across all the queries. All of the required BGPs in this portion of the query are joined upon the subject. As all the predicates and objects for a certain subject are available on the same line of input, the data can be joined in the Map stage. A further optimisation implemented here is the collection of all the required values for the ?range portion of the query in the same job. The current Jena implementation would recompute the ?range joins each time for the minimum and maximum queries, even though the difference is one BGP. In the Hadoop implementation, both the minimum and the maximum values are collected in the same job. After the values have been joined together they are collected and then written onto separate files in the HDFS via a special output function called Hadoop Multiple Outputs. By default, any output from a Map/Reduce task is combined into a single file large file. However using Hadoop Multiple Output enables the output from any task to be split into separate files, but predetermined files, when being stored on the HDFS [9]. The procedure described above is used for other triples groups which meet the requirements to be joined in the map phase. Each separate triple group is written out to its own unique file. This is done so that each triple group can be accessed directly at a later time, avoiding a costly re-search for them. The location of this process, along with the other triples groups which qualify from a map-side join can be seen in figure 4. The figure shows how the various triple groups joined via a map-side join are collected and written into separate files on the HDFS using Hadoop Multiple Outputs. This step is completed without the need to initiate a reduce task.

For joins that cannot be performed via a map-side join, they must be completed via a reduce-side join. An example case for using a reduce-side join would be when an object from one triple must be joined to a subject from a second triple. An example of this is shown in figure 3.

To join the triples shown in figure 3 via a reduce-side join, two stages must be performed. Firstly in the map phase, once one of the triples to be joined has been located, the element on which it is to be joined is set as the key and the rest of the triple as the value. In addition, the value is tagged with its join group and its join order. Any common elements will be passed to the same reducer. The join group is used in the reduce stage to decide which elements are members of which triple groups. The join order is used to determine a set order for the elements when being written out on the HDFS. Figure 4 shows the location of the reduce-side join. It also shows how the final output from any reduce-side joins are also written back onto the HDFS using the Hadoop Multiple Outputs feature. This is done so that only the files which contain the relevant triples can be used as input to the join stage, thus avoiding the need to rescan the entire contents of the HDFS.

4.2 Join Stage

This stage has two main functions. Firstly it performs the rest of the required joins to compete the queries. Secondly, it then formats and then emits the final output. This stage makes use of the broadcast-join method discussed earlier. The use of a broadcast-join enables numerous elements to be joined together from within the same in the same job. Using the existing methodology from the literature, the joins would otherwise have been performed via a series of costly cascade reduce-side joins. Figure 5 shows the workflow for the complete join stage Map/Reduce iteration.

To perform the broadcast-join, the smaller files created in the selection phase are distributed to all the reducers running in the join phase. This uniform distribution of files is achieved by making use of the Hadoop Distributed Cache feature. This feature allows smaller files to be made available to any Map or Reduce task running upon any node within the cluster. The files pushed to the reduce phase are extracted into Hashtables, which allows for extremely fast lookups when checking for element membership. The broadcast join method is particularly applicable in this case, since as highlighted in section 2.1, the data is massively skewed towards one triple group. This group, which always forms the ?obs part in any query, is far too large to be stored in memory so must be joined via a reduce-side join. However, the smaller triple groups can be joined to the larger ?obs group via numerous broadcast joins. The location of the broadcast join in the join stage workflow can be seen in figure 5.

These concepts are practically implemented in the following manner. Firstly the map phase, which takes as input a file containing the ?obs results from the reduce section. The map phase decides if the current input is an ?obs record or a ?a2 record based on length. These two triple groups are then joined via the common element and passed to the reduce phase. In the reduce phase the two groups which

will be available in the same job due the shuffle and sort phase. The system then performs the rest of the joins via the broadcast method. As explained above the files which are to be joined via the broadcast join method are all extracted from the Hadoop distributed cache and loaded into Hashtables in main memory. Then the different triple groups highlighted in figure 1 can be joined via any common elements. Each one of the joins performed via the broadcast method would otherwise have had to been completed via a separate reduce-side join, as they are all performed upon individual elements. Once the required joins are completed the algorithm then performs the conditional logic determined by original queries. The logic conditions include checking values against a pre-determined range. Figure 5 shows the full range of conditional logic check which the stage performs. Once the algorithm finds a value that does not meet the requirements, it will emit the required value and the other associated values back on the HDFS. This stage again makes use of the Hadoop Multiple Outputs, to split the output from each conditional logic check into its own file. This allows a user to more easily see and access the values which have failed a particular logic condition.

5. EXPERIMENTAL EVALUATION

5.1 Testing Methodology

The framework was tested on a dedicated Hadoop cluster. The cluster comprises a head node with eight data nodes. All of the machines are running CentOS 6.5 64-Bit, Java OpenJDK 1.7.0_51 and Hadoop 1.2.1. All the machines communicate via a dedicated Gigabit switch. The hardware specification of the cluster nodes is detailed in table 1. In this context a result is the time taken at the end of a successfully completed job, subtracted from the time at the start. These values are generated from within the Hadoop code itself and incorporate all the JVM initialisation and HDFS write stages. For the Jena-based results, the collection of the start and end time values as well as the running of the Jena binaries was performed by a shell script. All the experiments were repeated five times and an average taken to produce the final presented results.

5.2 Performance Scalability Across Number Of Nodes

To test how the Hadoop approach scales across different cluster sizes, it was run on different numbers of nodes. The results for 512M and 1000M are absent for the single node as the volume of storage space required to store these volumes of triples was greater than the 250GB of HDFS capacity available on the single node. Figure 6 shows how the upload algorithm scales across nodes. As the number of nodes increases, the time taken to complete the upload algorithm decreases. For example it took the single node cluster configuration 275 minutes to upload 256M triples, while it only took the eight node cluster configuration 30 minutes to process the same number of triples. Figure 7 shows how the query algorithm scales across different numbers of nodes. Again, increasing the number of nodes has a dramatic effect on the total time taken to complete the query algorithm. The increase in number of nodes always results in a decrease in total time taken to complete the collection of queries.

5.3 Query Stage Speed-Up

To assess the relationship between number of nodes and run time, a comparison of speed-up was performed. To create this comparison, the total time taken for the query algorithm of approach one across two, four and eight nodes was divided by the time taken for one node. This enables the speed-up resulting from increasing the number of nodes to become apparent. The best case scenario would be that the speed-up would reflect the increase in node numbers. For example, this best-case scenario would mean that time taken to complete a set task on one node would be eight times faster when running on eight nodes. Figure 8 shows the speed-up of different node numbers against a single node on dataset sizes up to 256M. The results highlight some very interesting trends in the query speed-up of approach one. The speed-up factor increases consistently as the number of nodes increases, which is to be expected. However the speed-up factor for a particular number of nodes does not stay consistent across dataset sizes. For example at 32M triples the speed-up factor of going from one node to eight nodes is 2, however at 256M triples the speed-up factor is 5.9.

5.4 Comparison With Jena TDB

To test how the Hadoop approach compares with Jena TDB, Jena was run on one of the nodes which comprises the Hadoop cluster. This allowed for direct comparisons to be drawn. The results shown for Jena TDB are the combination of the time taken for Jena to complete all the original eight SPARQL queries which the Hadoop approach is also using. The results show the combined upload and query time for the approaches to complete all eight queries. Figure 9 shows how the Hadoop approach, running on the full eight node cluster, compare against Jena running on a single node. The dataset size of 128M triples was the maximum size which Jena was able to successfully run against. This shows how effective the Hadoop approach to be when compared with Jena as it is over 22 times faster then Jena on a dataset size of 128M triples.

6. CONCLUSIONS

From such positive results when compared with Jena, it can said that Hadoop is very effective at storing and querying medical RDF data. The Hadoop approach demonstrates better performance then Jena when running on the same machine and processing the same datasets. The Hadoop approach also demonstrates scalability when tested in a distributed environment, meaning that it is well equipped to deal with NHS-sized datasets. Hadoop allows for a system to assess the quality of medical data which is not only better performing than using traditional Semantic Web technologies, but it is also able to process the massive volumes of data required by the NHS, as it scales the workload across a computer cluster. This project has also introduced two novel methods for use when completing SPARQL queries using map/reduce: the use of broadcast joins and the creation of the super-query.

The project uses the broadcast join method for completing many of the required joins. This method reduces the need for a series of cascade reduce-side joins. This project appears to be the first from the currently available literature to make use of the broadcast join method to perform queries on RDF data. This project also appears to be the first to introduce

the notion of the super-query. A super-query is created from a series of standard SPARQL queries and is used to save on the re-computation of common joins. This something neither Jena or the existing Hadoop-SPARQL approaches appear to implement.

The work performed for this paper will hopefully be used to allow for more and larger medical databases to be checked for errors and inconsistencies. This should hopefully lead to more accurate and reliable databases being used in both medical research and also for diagnosis.

7. ACKNOWLEDGMENTS

The authors would like to acknowledge the work of the BrainIT group investigators and participating centres to the BrainIT dataset. The authors would also like to acknowledge the use of the University of Huddersfield Queensgate Grid in carrying out this work

8. REFERENCES

[1] D. Agrawal, P. Bernstein, E. Bertino, S. Davidson, U. Dayal, M. Franklin, J. Gehrke, L. Haas, A. Halevy, J. Han, H. V. Jagadish, A. Labrinidis, S. Madden, Y. Papakonstantinou, J. Patel, R. Ramakrishnan, K. Ross, C. Shahabi, D. Suciu, S. Vaithyanathan, and J. Widom. Challenges and Opportunities with Big Data 2011-1. 2011.

[2] I. Chambers, B. Gregson, G. Citerio, P. Enblad, T. Howells, K. Kiening, J. Mattern, P. Nilsson, I. Piper, A. Ragauskas, et al. Brainit collaborative network: analyses from a high time-resolution dataset of head injured patients. In *Acta Neurochirurgica Supplements*, pages 223–227. Springer, 2009.

[3] D. Corsar, L. Moss, and I. Piper. Data quality assessment using linked data: A case study in the medical domain. In *E-KAW 2012, The 18th International Conference on Knowledge Engineering and Knowledge Management*, 2012.

[4] B. Feldman, E. M. Martin, and T. Skotnes. Big data in healthcare hype and hope. 2012.

[5] S. I. Goldberg, A. Niemierko, and A. Turchin. Analysis of data errors in clinical research databases. In *AMIA Annual Symposium Proceedings*, volume 2008, page 242. American Medical Informatics Association, 2008.

[6] D. Howe, M. Costanzo, P. Fey, T. Gojobori, L. Hannick, W. Hide, D. P. Hill, R. Kania, M. Schaeffer, S. St Pierre, et al. Big data: The future of biocuration. *Nature*, 455(7209):47–50, 2008.

[7] L. Moss, D. Corsar, and I. Piper. A linked data approach to assessing medical data. In P. Soda and F. Tortorella, editors, *25th International Symposium on Computer-Based Medical Systems (CBMS), 2012*, pages 1–4, 2012.

[8] K. Rohloff and R. E. Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: the shard triple-store. In E. Tilevich and P. Eugster, editors, *PSI EtA*. ACM, 2010.

[9] T. White. *Hadoop: The Definitive Guide*. O'Reilly, second edition edition, 2010.

APPENDIX

A. TABLES

A.1 Cluster Specification

Component	Head Node	Data Node
CPU	Intel Q8400	Intel Q8400
RAM	4GB DDR2	8GB DDR2
HDD	250GB (7200RPM)	250GB (7200RPM)

Table 1: Specification Of The Hadoop Cluster

A.2 Subject, Predicate and Object Distribution

RDF Element	Number Of Elements
Subject	1,523,106
Predicate	61
Object	1,711,702

Table 2: Subject, Predicate and Object Distribution

A.3 Triple Group Distribution

Triple Group Name	Number Of Elements
?obs	3,426,188
?range	518
?cs	446

Table 3: Triple Group Distribution

B. DIAGRAMS

B.1 Super-Query

```
?range a med:AcceptableRange .
?range med:clinicalRangeMax ?max .
?range med:clinicalRangeMin ?min .
?range pd:hasParameter ?p .

?obs a mo:PhysiologicalObservation .
?obs ssn:observedProperty ?p .
?obs ssn:observationResultTime ?time .
?obs pd:atHumanTime ?htime .
?obs ssn:observationResult ?a1
?obs ssn:observedBy ?sensor.

?a1 ssn:hasValue ?a2 .
?a2 pd:readingValue ?value .

?sensor ssn:hasMeasurementCapability ?mc.

?mc a ssn:Accuracy .
?mc ms:capabilityValue ?accuracy .

med:Hypertension med:requiredSymptoms ?cs .
med:Hypotension med:requiredSymptoms ?cs .
```

```
?cs med:clinicalFeatures ?cscf .

?cscf pd:hasParameter ?p .
?cscf med:clinicalRangeMax ?csrMax .
?cscf med:clinicalRangeMin ?csrMin .
```

B.2 Triples Groups

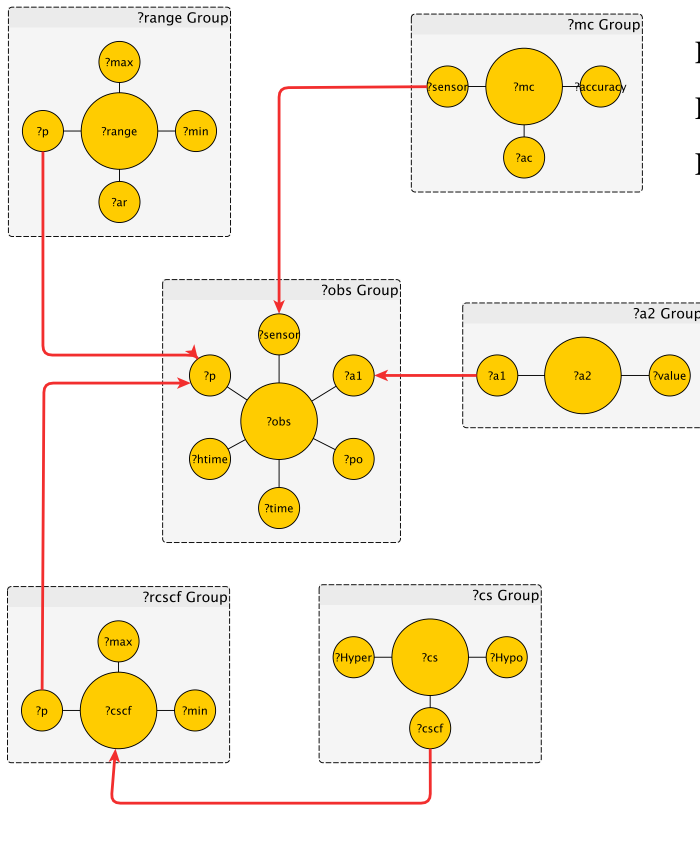

Figure 1: Triple Group Joining Plan

B.3 Map-Side Join Example

```
?range a med:AcceptableRange.
?range med:clinicalRangeMax ?max.
?range pd:hasParameter ?p.
```

Figure 2: Example Section Of A SPARQL Query That Can Be Joined Via A Map-Side Join

B.4 Reduce-Side Join Example

B.5 Selection Stage

B.6 Join Stage

B.7 Upload Performance Scalability Across Nodes

```
?a1 ssn:hasValue ?a2.
?a2 pd:readingValue ?value.
```

Figure 3: Example Section Of A SPARQL Query That Can Be Joined Via A Reduce-Side Join

B.8 Query Performance Scalability Across Nodes

B.9 Query Speed-Up Factor

B.10 Hadoop Approach Comparison With Jena

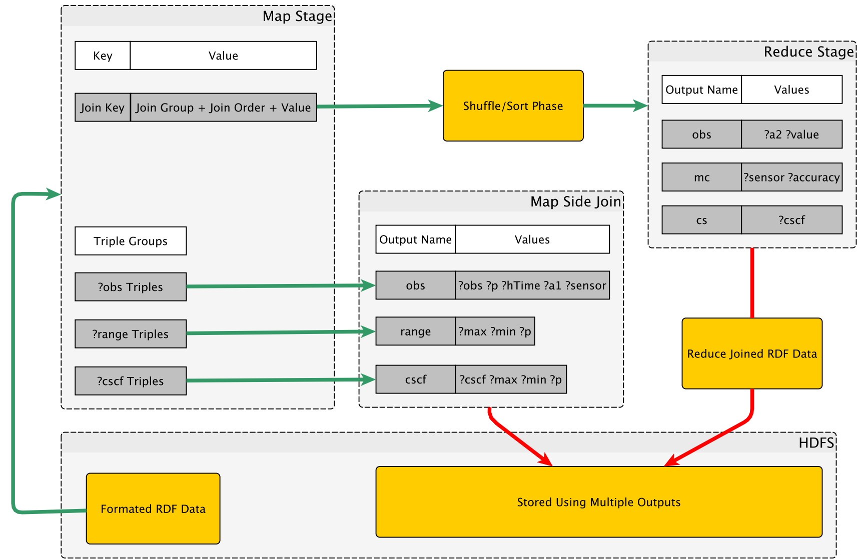

Figure 4: Selection Stage Workflow

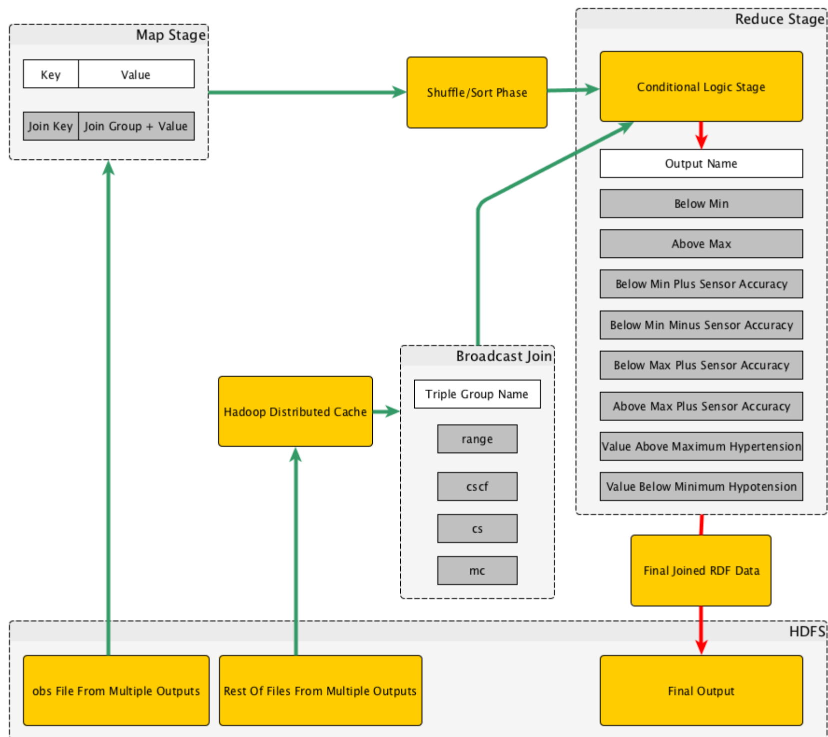

Figure 5: Join Stage Workflow

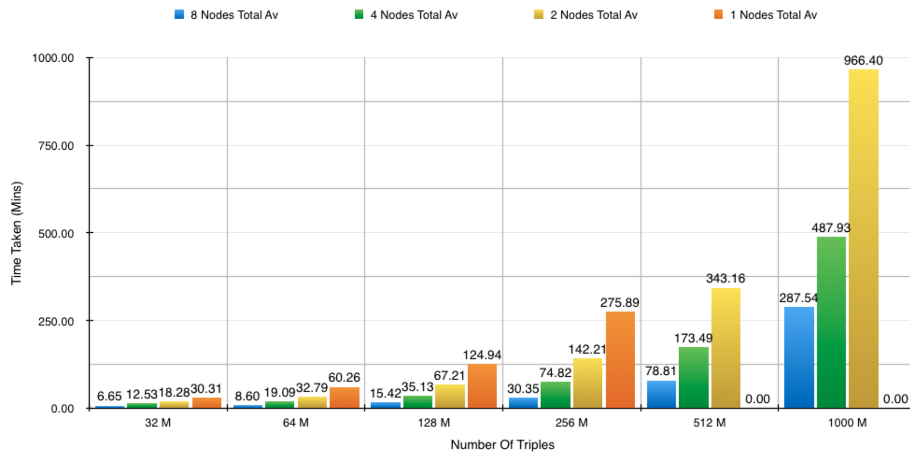

Figure 6: Upload Performance Scalability Across Nodes

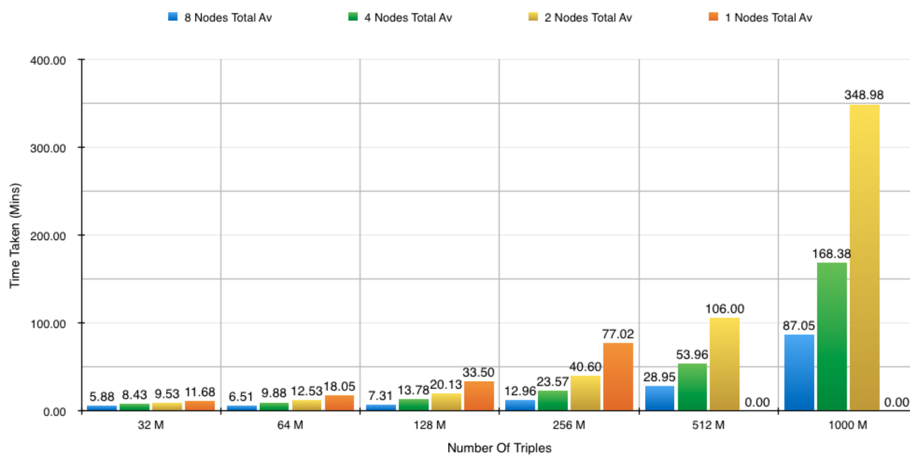

Figure 7: Query Performance Scalability Across Nodes

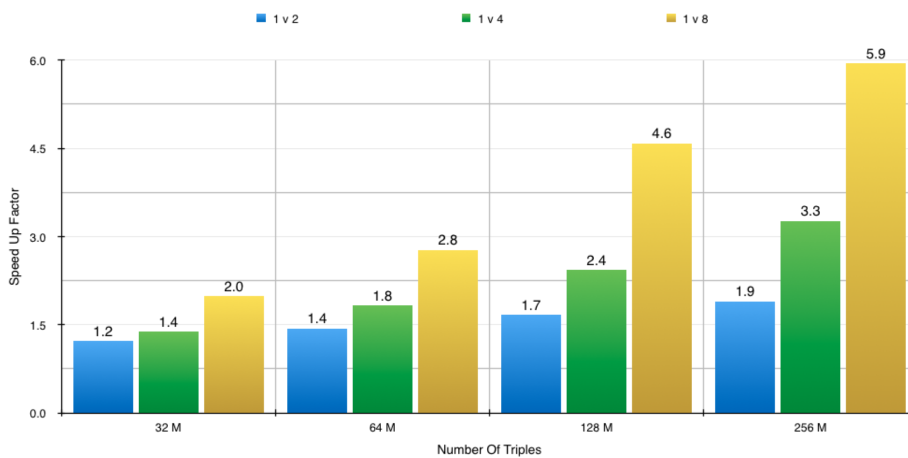

Figure 8: Query Speed-Up Factor

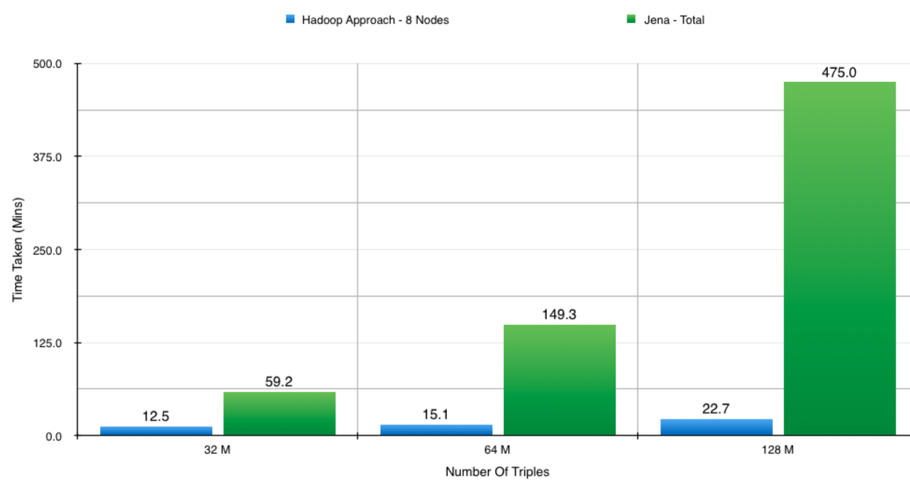

Figure 9: Hadoop Approach Comparison With Jena