2013 IEEE Seventh International Conference on Semantic Computing

Analysis of Big Data Technologies and Methods

- Query Large Web Public RDF Datasets on Amazon Cloud Using Hadoop and Open Source Parsers

Ted Garcia and Taehyung ("George") Wang
Department of Computer Science
California State University, Northridge
Northridge California, 91330
{ted.garcia.574, twang}@csun.edu

Abstract - **Extremely large datasets found in Big Data projects are difficult to work with using conventional databases, statistical software, and visualization tools. Massively parallel software, such as Hadoop, running on tens, hundreds, or even thousands of servers is more suitable for Big Data challenges. Additionally, in order to achieve the highest performance when querying large datasets, it is necessary to work these datasets at rest without preprocessing or moving them into a repository. Therefore, this work will analyze tools and techniques to overcome working with large datasets at rest. Parsing and querying will be done on the raw dataset - the untouched Web Data Commons RDF files. Web Data Commons comprises five billion pages of web pages crawled from the Internet. This work will analyze available tools and appropriate methods to assist the Big Data developer in working with these extremely large, semantic RDF datasets. Hadoop, open source parsers, and Amazon Cloud services will be used to data mine these files. In order to assist in further discovery, recommendations for future research will be included.**

Keywords-big data; Hadoop; Amazon cloud computing; RDF; Jena; NXParser; Any23; Semantic Web; Map/Reduce; open source software

I. INTRODUCTION

Querying large datasets has become easier with Big Data technologies such as Hadoop's MapReduce. Large public datasets are becoming more available and can be found on the Amazon Web Service (AWS) Cloud. In particular, Web Data Commons [1] has extracted and posted RDF Quads from the Common Crawl Corpus [2] found on AWS [3] which comprises over five billion web pages of the Internet. Technologies and methods are in their infancy when attempting to process and query these large web RDF datasets. For example, within the last couple of years, AWS and Elastic MapReduce (EMR) [4] have provided processing of large files with parallelization and a distributed file system. RDF technologies and methods have existed for some time and the tools are available commercially and open source. RDF Parsers and databases are being used successfully with moderately sized datasets. However, the use and analysis of RDF tools against large datasets, especially in a distributed environment, is relatively new. In order to assist the BigData developer, this work explores several open source parsing tools and how they perform in Hadoop on the Amazon Cloud. Apache Any23 [5] Apache Jena RIOT/ARQ [6], and SemanticWeb's NxParser [7] are open source parsers that can process the RDF quads contained in the Web Data Commons files. It should be known that because these parsers support limited query functions, this work will analyze extract and parse functionality only. Therefore, more comprehensive tools should be used or included for complex queries.

Currently, using conventional databases and other software tools have not proven efficient when working with large datasets. Therefore, we suggest working with these datasets at rest without preprocessing or importing them into a repository. In order to address this challenge this work will show how current RDF parsers can be used with Hadoop running on Amazon Cloud services. Furthermore, suggestions will be given on approaches and methods to allow the best use of these parsers and technologies for a variety of scenarios.

In Section 2, we introduce big data technologies along with Amazon cloud technologies, Hadoop Map/Reduce, and Parsers. In Section 3, we describe related works that include NQuads, SPARQL, SimpleDB, and Web Data Commons. Section 4 discusses the analysis and design of a query program developed for this study that will compare performance of the three open source parsers. In Section 5, we present the analysis of the results of testing. Finally, the conclusion and discussion regarding future research is addressed in Section 6.

II. BIG DATA TECHNOLOGIES

A 2011 McKinsey paper suggests suitable technologies include A/B testing, association rule learning, classification, cluster analysis, crowd sourcing, data fusion and integration, ensemble learning, genetic algorithms, machine learning, natural language processing, neural networks, pattern recognition, predictive modeling, regression, sentiment analysis, signal processing, supervised and unsupervised learning, simulation, time series analysis and visualization. Additional technologies being applied to big data include massively parallel-processing (MPP) databases, search-based applications, data-mining grids, distributed file systems, distributed databases, cloud computing platforms, the Internet, and scalable storage systems [8].

Even though there are many suitable technologies to solve Big Data challenges as indicated in the list above given by the McKinsey paper, this work will explore a handful of the more popular ones, in particular the Amazon Cloud, Hadoop's MapReduce, and three open-source parsers.

A. *Amazon Cloud Technologies*

Amazon Web Services (AWS) cloud platform, is one of today's most popular and is shown in Fig. 1. It already has Hadoop's MapReduce Framework available for use. We

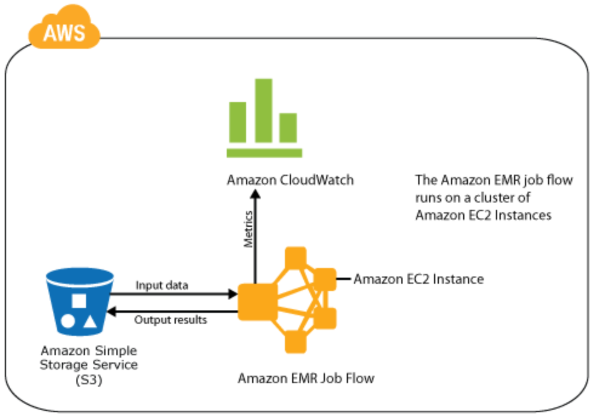
AWS
Amazon CloudWatch
The Amazon EMR job flow runs on a cluster of Amazon EC2 Instances
Metrics
Input data
Output results
Amazon EC2 Instance
Amazon Simple Storage Service (S3)
Amazon EMR Job Flow

Figure 1. Amazon Services Architecture [4].

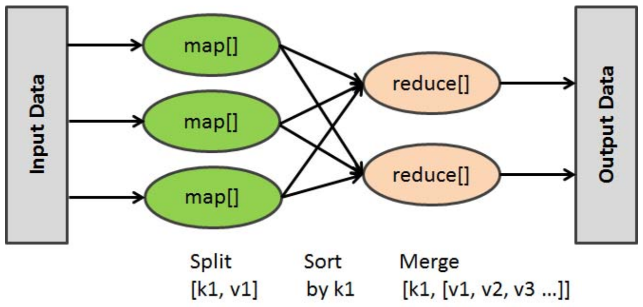

Figure 2. Map/Reduce Directed Acyclic Graph [19].

store the large RDF files in Amazon Simple Storage Service (S3) [9] and use the Amazon Elastic MapReduce (EMR) [4] framework running on the Amazon Elastic Compute Cloud (EC2) [3] to run the parsers.

B. *Hadoop's Map/Reduce*

The Hadoop Map/Reduce architecture provides a parallel processing framework that might be a better solution than multithreading. Multithreading requires adept knowledge and skill for the programmer in that coordinating each thread and critical section can cause many problems. Multithreading requires semaphores, locks, etc. which require tedious testing to ensure there is no race or deadlock conditions. Hadoop eliminates the shared state completely and reduces the issues mentioned above.

This is the fundamental concept of functional programming. Data is explicitly passed between functions as parameters or return values which can only be changed by the active function at that moment. In this case functions are connected to each other in the shape of a directed acyclic graph. Since there is no hidden dependency (via shared state), functions in the directed acyclic graph can run anywhere in parallel as long as one is not an ancestor of the other.

Map/reduce is a specialized directed acyclic graph which can be used for many purposes [19]. It is organized as a “map” function which transforms a piece of data into some number of key/value pairs. Each of these elements will then be sorted by their key and reach to the same node, where a “reduce” function is used to merge the values (of the same key) into a single result as shown in Fig. 2. The code snippet below shows typical map and reduce functions. Several may be chained together to implement a parallel algorithm for different use cases.

```
map(input_record) {
     ...
     emit(k1, v1)
     ...
     emit(k2, v2)
     ...
}
reduce (key, values) {
     aggregate = initialize()
     while (values.has_next) {
          aggregate = merge(values.next)
     }
     collect(key, aggregate)
}
```

C. *Hadoop Distributed File System (HDFS)*

The distributed file system is another innovation essential to the performance of the Hadoop framework. HDFS can handle large files in the gigabytes and beyond with sequential read/write operation. These large files are broken into chunks and stored across multiple data nodes as local files.

There is a master “NameNode” to keep track of overall file directory structure and the placement of chunks. This NameNode is the central control point and may re-distribute replicas as needed. DataNode works all its chunks to the NameNode at bootup.

To read a file, the client API will calculate the chunk index based on the offset of the file pointer and make a request to the NameNode. The NameNode will reply which DataNodes has a copy of that chunk. From this point, the client contacts the DataNode directly without going through the NameNode.

To write a file, client API will first contact the NameNode who will designate one of the replicas as the primary (by granting it a lease). The response of the NameNode contains who is the primary and who are the secondary replicas. Then the client pushes its changes to all DataNodes in any order, but this change is stored in a buffer of each DataNode. After changes are buffered at all DataNodes, the client sends a “commit” request to the primary, which determines an order to update and then pushes this order to all other secondaries. After all secondaries complete the commit, the primary will respond to the client about the success. Fig. 3 shows HDFS architecture.

D. *Parsers*

The Web Data Commons files are stored in RDF NQuads [10] and can only be parsed directly with a small handful of parsers. Triples, the more popular format is supported by almost all RDF parsers and tools.

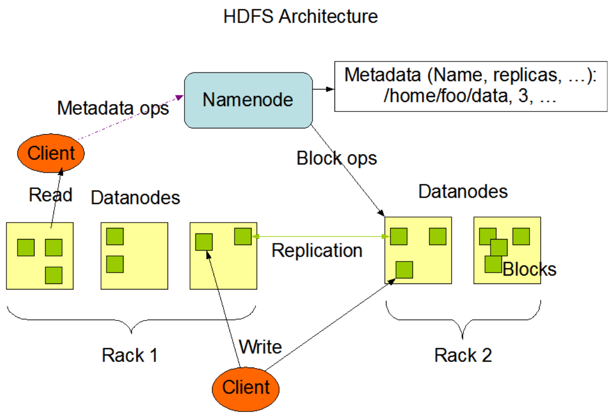

Figure 3. HDFS Architecture (Apache Hadoop) [20].

Apache Jena ARQ/RIOT is a Java framework for building Semantic Web applications. Jena provides a collection of tools and Java libraries to help develop semantic web and linked-data applications, tools and servers.

The Jena Framework includes:

- an API for reading, processing and writing RDF data in XML, N-triples and Turtle formats - this work is using this module.
- An ontology API for handling OWL and RDFS ontologies.
- A rule-based inference engine for reasoning with RDF and OWL data sources.
- Stores to allow large numbers of RDF triples to be efficiently stored on disk.
- A query engine compliant with the latest SPARQL specification.
- Servers to allow RDF data to be published to other applications using a variety of protocols, including SPARQL.

In April 2012, Jena graduated from the Apache incubator process and was approved as a top-level Apache project [6].

The Jena project has contributed extensively to the RDF movement including the following important principle. "RDF has an XML syntax and many who are familiar with XML will think of RDF in terms of that syntax. This is a mistake. RDF should be understood in terms of its data model. RDF data can be represented in XML, but understanding the syntax is secondary to understanding the data model" [11].

Anything To Triples (Any23) [5] is a library, a web service and a command line tool that extracts structured data in RDF format from a variety of Web documents.

- RDF/XML, Turtle, Notation 3.
- RDFa with RDFa 1.1 prefix mechanism.
- Microformats: Adr, Geo, hCalendar, hCard, hListing, hResume, hReview, License, XFN and Species.
- HTML5 Microdata: (such as Schema.org).
- CSV: Comma Separated Values with separator auto-detection.

Apache Any23 is used in major Web of Data applications such as sindice.com and sigma. It is written in Java and licensed under the Apache License. Apache Any23 can be used in various ways:

- As a library in Java applications that consume structured data from the Web.
- As a command-line tool for extracting and converting between the supported formats.
- As online service API available at any23.org.

Any23 is based on the Sesame Parser and appears to have ended its incubator status in August of 2012. However, at this time, it is not listed in the formal list of Apache Projects [12].

NxParser is a Java open source, streaming, non-validating parser for the Nx format, where x = Triples, Quads, or any other quantity. For more details see the specification for the NQuads format, an extension of the N-Triples RDF format. Note that this parser handles any combination or number of N-Triples syntax terms on each line (the number of terms per line can also vary).

NxParser performs very well. The following was indicated on their website. It ate 2 million quads (~4GB, (~240MB compressed)) on a T60p (Win7, 2.16 GHz) in approximately 1 minute and 35 seconds. Overall, it's more than twice as fast as the previous version when it comes to reading Nx.

It comes in two versions: lite and not-so-lite. The latter is provided "as-is" and comes with many features, most of which is not needed for this work. This work used the lite version. There is some code for batch sorting large-files and various other utilities, which some may find useful. If you just want to parse Nx into memory, use the lite version.

The NxParser is non-validating, meaning that it will happily eat non-conformant N-Triples. Also, the NxParser will not parse certain valid N-Triples files where (i) terms are delimited with tabs and not spaces; (ii) the final full-stop is not preceded by a space [7]. The NxParser comes from YARS which was created by DERI Galway.

DERI Galway, the Digitial Enterprise Research Institute is an institute of the National University of Ireland, Galway which is located in Galway, Ireland. The focus of the research in DERI Galway is on the Semantic Web [13].

YARS stands for "Yet another RDF store". YARS is a data store for RDF in Java and allows for querying RDF based on a declarative query language, which offers a somewhat higher abstraction layer than the APIs of RDF toolkits such as Jena or Redland. YARS uses Notation3 as a way of encoding facts and queries [14].

III. Related Works

This work is unique. Many days of research were spent looking for exactly the same kind of work. The IEEE was searched as well as the Internet but nothing was found that deals with these three parsers in Amazon Hadoop.

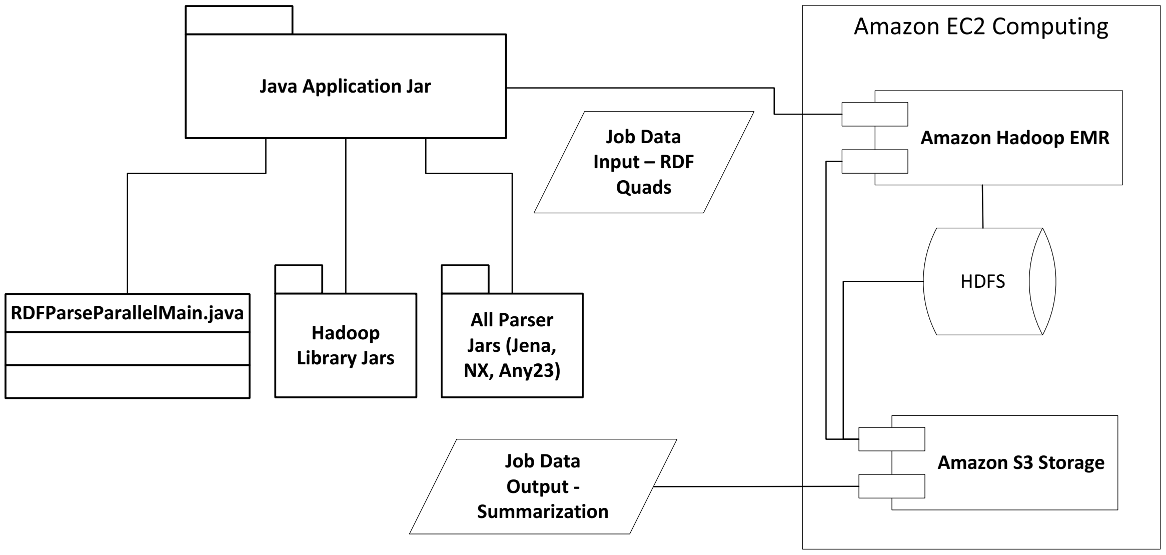
Java Application Jar
RDFParseParallelMain.java
Hadoop Library Jars
All Parser Jars (Jena, NX, Any23)
Job Data Input – RDF Quads
Job Data Output - Summarization
Amazon EC2 Computing
Amazon Hadoop EMR
HDFS
Amazon S3 Storage

Figure 4. Query Program Architecture.

Furthermore, there are not many RDF benchmarks for BigData technologies. Let this work be one of the first. Even though this work is unique, the following three studies have similarities to it.

The Data Intensive Query Processing for Large RDF Graphs Using Cloud Computing [15] work processed RDF with Hadoop in the Cloud like this work but it used a database and triples instead of no database and NQuads. Mohammad's work is also not concerned with the Web Data Commons content containing most of the Internets pages of RDF. Mohammad's work described an algorithm that determines the performance cost of executing a SPARQL query on the RDF. The RDF query language SPARQL is used to query an RDF repository. In Mohammad's work, RDF needs to be converted and it is handed over to the Hadoop cluster to process. In some ways Mohammad's work is a performance analysis of an algorithm that uses Hadoop. On the other hand, this work compares the performance of three parsers being testing on the Amazon application services.

The RDF on Cloud Number Nine [16] work used SimpleDB on the Amazon Cloud. The team created an RDF store which acts as a back end for the Jena Semantic Web framework and stores its data within the SimpleDB. They stored RDF triples and queried them while checking performance against the Berlin SPARQL Benchmark to evaluate their solution and compare it to other state of the art triple stores. This work is a great example of using Jena and the Amazon Cloud to query RDF. However, this work, like the one above, use databases to store and process RDF. There are still challenges using databases when data sizes are in gigabytes, terabytes, and beyond that these other studies do not address.

The RDF Data Management in the Amazon Cloud [17] work also used SimpleDB in the Amazon Cloud. This team attempted to use various indexing techniques to try to guess the query path in order to optimize performance.

Finally, the Web Data Commons – Extracting Structured Data from Two Large Web Corpora [18] worked with the content of the entire Internet represented by RDF Quads and showed the popularity of various format types found within the pages.

IV. Analysis and Design of the Query Program

This section will cover the design of the program that compares performance of the three open source parsers by querying large RDF data sets. Included will be an overview of the process and methods of performing such work on the Amazon Cloud using Java technology. This section will describe the components of the system used to compare the parsers as shown in Fig. 4 and Table I. This work will query the Web Data Common RDF extraction from the Common Crawl Corpus which comprises over five billion web pages on the Internet. The query will be performed by three open source parsers running separately on Hadoop in the Amazon Cloud.

TABLE I. Component Descriptions

Component	Description
RDF Files	Web Data Commons' RDF NQuad files
Amazon EC2	Highly scalable, parallel computing infrastructure
Amazon S3	Scalable and distributed storage running on EC2
Amazon EMR	Implementation of Hadoop's MapReduce services running on EC2
Parsers	Jena, NXParser, and Any23 open source parser programs used to manipulate the RDF NQuad files
Java	Language used to write the query program with Hadoop libraries that compares parser performance

They are run separately to compare their performance extracting and parsing the RDF NQuad files. This will allow us to perform analytics on the entire Internet by scanning the RDF files and performing query functions on the NQuads. The high level process is described below:

A. *Process Steps*

The basic steps that were conducted to obtain the results of this work are described as follows:

1) Start with raw web pages containing RDFa from the Common Crawl Corpus.
2) Obtain compressed RDF NQuads converted from RDFa of the Common Crawl Corpus - done by Web Data Commons (WDC).
3) Decompress WDC files and store them on Amazon S3- done by this work using standard Amazon file utilities.
4) Run Amazon Elastic Map Reduce (Hadoop) and parsers on decompressed files using this works program: RDFParseParallelMain.java.
5) Combine Hadoop's distributed analytics output files into one file and create graphs of results.

How Hadoop was used:

1) Determine the number of jobs needed to answer a query.
2) Minimize the size of intermediate files so that data copying and network data transfer is reduced.
3) Determine number of reducers.

Note that usually we run one or more MapReduce jobs to answer one query. We use the map phase to select data and the reduce phase to group it.

B. *Common Crawl Corpus*

Common Crawl is an attempt to create an open and accessible crawl of the web. Common Crawl is a Web Scale crawl, and as such, each version of the crawl contains billions of documents from the various sites that were successfully able to crawl. This dataset can be tens of terabytes in size, making transfer of the crawl to interested third parties costly and impractical. In addition to this, performing data processing operations on a dataset this large requires parallel processing techniques, and a potentially large computer cluster. [2] For this reason, we used the RDF subset created by Web Data Commons.

C. *Web Data Commons (WDC)*

More and more websites embed structured data describing products, people, organizations, places, events, resumes, and cooking recipes into their HTML pages using markup formats such as RDFa, Microdata and Microformats. The Web Data Commons project extracted all Microformat, Microdata and RDFa data from the Common Crawl web corpus. The corpus is the largest and most up-to-date web corpus that is currently available to the public. The corpus provides the extracted data for download in the form of RDF-quads and also in the form of CSV-tables for common entity types (e.g. product, organization, location, etc.). In addition, Web Data Commons calculates and publishes statistics about the deployment of the different formats as well as the vocabularies that are used together with each format.

Web Data Commons has extracted all RDFa, Microdata and Microformats data from the August 2012 and the 2009/2010 Common Crawl corpus. This work will use the August 2012 RDF files which is about 100 gigabytes. Web pages are included into the Common Crawl corpora based on their page rank score, thereby making the crawl's snapshots of the current popular part of the Web. For the future, Web Data Commons plans to rerun their extraction on a regular basis as new Common Crawl corpora are becoming available [1].

D. *Amazon Cloud*

This work will use several services on the Amazon Cloud. The Hadoop framework implemented by Amazon Elastic Map Reduce (EMR) will be used to analyze the large set of RDF data. The EMR provides fine-grain metrics of many aspects of the running job. From these metrics, we produce the test results charts. Since WDC has done some of the difficult work for us, we start with fairly structured and validated RDF NQuad files. These files are stored in the Amazon S3.

After "jarring" up the RDFParseParallelMain program with the parsers jar files, this final jar file is saved to the S3 as well. We also setup an output directory for the query result files.

To run RDFParseParallelMain we used the EMR control panel and start a new job. The EMR will run the RDFParseParallelMain program on as many processors as we indicate and write output files to the S3.

E. *RDFParseParallelMain*

The capability of the program RDFParseParallelMain is to twofold: count entries for each node type in the NQuad, i.e. subject, predicate, object, or context or count occurrences of a query string in a specified node type. The parser comparison testing was done using the former capability of counting occurrences of the node type. We have provided extensive argument passing so we could run the tests in various modes using each of the parsers separately.

The map and reduce functions are fairly simple. The map for the testing is key = node type; value = count. The map for the query is key = query string; value = count. The reduce function simply adds the reoccurring words and outputs the total to the output file. The complexity and main processing in this program is in the parsers. Each parser must break down the NQuad into separate nodes. Once that is done, the program passes the node to the reduce function where it eventually gets counted.

The final Java jar file was a jar of jars along with the main Java program. In other words, the Hadoop, Jena, NXParser, and Any23 application jar files, several for each, were all "jarred" together with the RDFParseParallelMain Java program. This composite jar file is what is used by the Amazon EMR to run the tests.

F. Test Design

The testing of each parser is done separately. This ensures there is sufficient control to yield more accurate performance results. The RDFParseParallelMain program was written such that no processing is done within it. Instead, all processing was handed over to the parsers. The RDFParseParallelMain only provided transport between the input and the Hadoop "reduce" method. More specifically, the Any23 and Jena parsers used callbacks to process whereas the NXParser parser used a tight loop.

The RDF NQuad files sizes were 1, 4, 16, and 24 gigabytes and the CPU count is 10, 24, 64, and 90 CPU's. These quantities are not entirely random. Earlier testing, not documented in this work, gave a basis to establish amounts for this testing. The file sizes were established based on a somewhat smooth progression. The last file size, which ideally would have been 32, of 24 was used to reduce the cost of processing and uploading the files. The CPU count was established with similar justifications since Amazon has a significant cost increase over 100 CPU's. Unfortunately, cost became an issue in determining how much of the Amazon processing power would be reasonable for this project. But it appears these constraints did not deter from a good understanding of the parsers' behaviors.

In the next section, we will analyze the test results and what has been concluded. Hopefully, the results and testing methodology will be useful to other Big Data and Semantic Web practitioners.

V. Results

We will analyze two aspects of the results, i.e. the performance of the parsers and the CPU cost of each parser. The performance comparison will include looking at the performance curve to see where each parser did their best. For example, parser X did its best when processing the 4.7 GB moderately sized file with 64 CPU's. In the same way, we can look at the CPU cost curve and see where the best CPU cost occurs.

Before looking at the results, it is important to understand a few points. Statistical integrity was not achieved in this testing due to the cost to run the jobs. However, due to the stability and consistency of the Amazon Cloud infrastructure and how the tests were performed, there is significant value in the findings. The main question is, if we rerun the tests again will we get significantly different results. As mentioned before, prior tests were conducted and they show minimal variance. Therefore, for this experiment, outcomes of the testing are taken with full face value, even though they are not statistically proven. We have suggested in the future work section to run more tests for statistical validity.

In general, the testing produced unexpected results. We expected to see normal patterns in the charts. Normal curves did not show in the results. However, the results seem to have some consistency within the overall testing. In other words, the charts show unusual behavior of many of the tests but there seems to be reasonable justification. Each parser seems to respond to the CPU and file size combinations in differing ways. This may suggest that the parsers can be used optimally under certain scenarios.

Fig. 5 allows us to compare and contrast performance among the parsers. For the most part, Jena is the slowest overall. Any23 and NXParser out-performed each other in different scenarios of file size and CPU count. For the important test where the largest size of file meets low CPU count and therefore CPU utilization, NXParser seems to have edged out the others.

If we look at each file size and then the CPU count we can see that there is a different winner for each scenario. For the 0.7 GB file the NXParser parser was the fastest in the extreme cases of 10 CPU's and 90 CPU's. It was the slowest in the middle CPU counts. Then the NXParser parser was the fastest in the 24 and 90 runs of the 4.7 GB file and slowest on 64 and the middle on 10. For the final 15.9 GB file, the NXParser parser was the fastest at 10 and 24 CPU's.

In the following, we explore each of the parsers separately and their individualized performance curves. As mentioned before, each parser does better or worse at a particular file size to CPU count permutation. We have taken the parser performance chart (Fig. 5) and created separate charts for each parser to ensure the individualization is apparent.

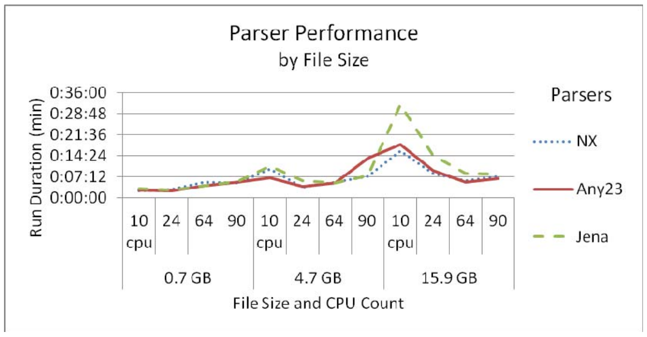

Figure 5. Parser Performance.

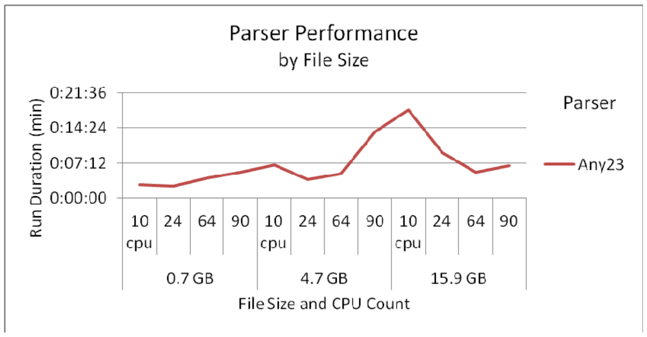

Figure 6. Parser Performance - Any23.

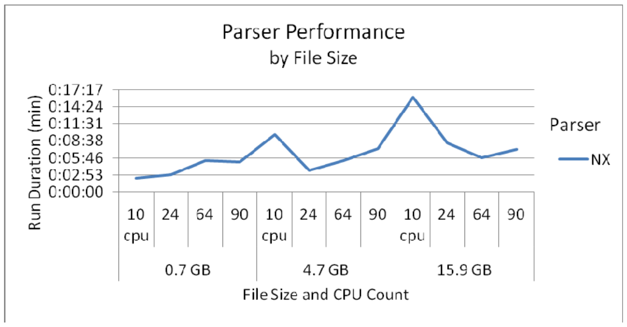

Figure 7. Parser Performance - NXParser.

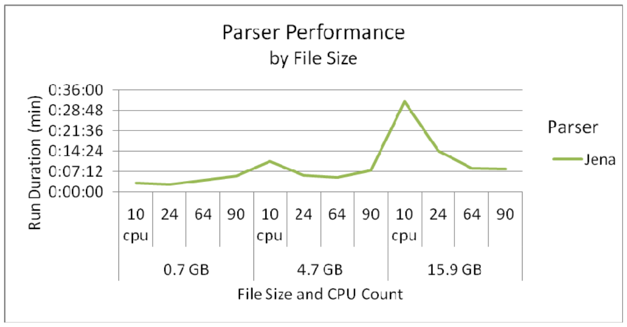

Figure 8. Parser Performance - Jena.

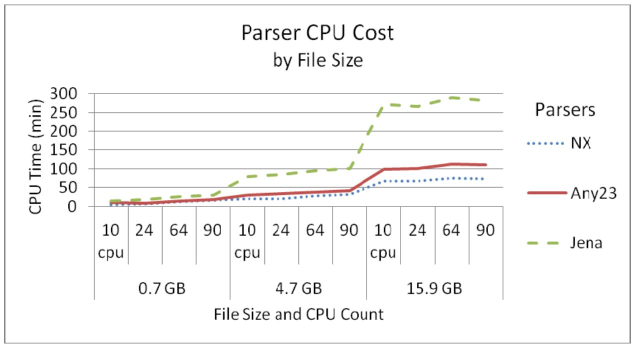

Figure 9. Parser CPU Cost by File Size.

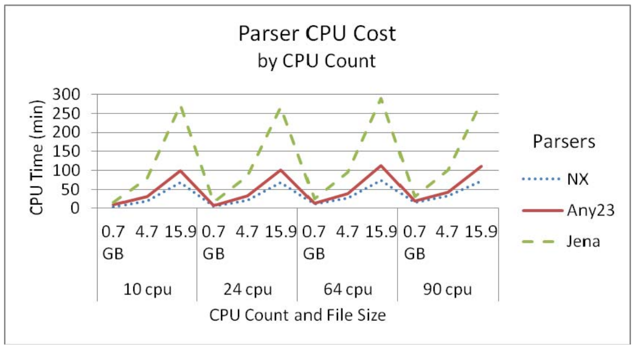

Figure 10. Parser CPU Cost by File Size.

The Any23 parser as shown in Fig. 6 was in the middle of almost all permutations except it was faster in the 4.7 GB, 10 CPU and 15.9 GB, 64 and 90 CPU runs. Strangely, it was much slower in the 4.7 GB, 90 CPU run. For the smallest file it is slow at first, gains momentum and then quickly slows down again. Apparently the overhead of CPU's slows it down. For the medium file size, it optimizes performance at the 24 CPU count. For the large file, it improves over the initial progression of CPU count but falters slightly at the 90 CPU count.

The NXParser as shown in Fig. 7 and Any23 parser had very similar performance characteristics which make it difficult to see which parser would work better for particular scenarios. It appears the NXParser parser was faster more times than Any23. The NXParser also has the best CPU usage index as noted in the next section. The NXParser shows normal time increases for the small file but speeds up when using 90 CPU's. The optimal time on the medium size file is 24 CPU's and does not take advantage of more CPU's. The large file performance improves as expected with more CPU's until the use of 90 CPU's where there is a minor dip.

The Jena parser as shown in Fig. 8, for the most part, was the slowest of all and used the most CPU time, especially in the 15.9 GB file runs. Oddly, it was the best performer in the 0.7 GB, 64 CPU run. For the small file it progressed negatively as the CPU count increased. On the medium file size, performance increased as the number of CPU's increased until the 90 CPU count. Finally, Jena took full advantage of throwing CPU's at it.

It is not always enough to see how fast a program runs but also the cost of running it. Especially when running on cloud environments where there is a fee for everything and costs can grow quickly. The next section compares the three parsers CPU usage and therefore the cost to run the parser and are shown in Fig. 9.

The Parser CPU Cost by File Size chart, shown in Fig. 10, correlates with the Parser Performance chart. It shows the total time spent running the job by adding together the CPU utilization for each of the CPU's. The Fig. 9 chart shows that the parsers CPU usage is fairly close for the smaller file sizes but not for the larger files. There seems to be a dip at the 90 CPU test for all the parsers. The dip correlates to the performance graph of each of the processors counts. This may indicate there is a diminishing return as the CPU's increase for each file size.

Reviewing both charts it is very clear that Jena utilizes the most CPU than the other two. The NXParser seems to be the least greedy when utilizing compute resources. Therefore the NXParser would be the least expensive to run in most cases.

Fig 10. shows CPU utilization per parser based on different number of processors. This last chart shows CPU utilization from a slightly different perspective. The chart is similar to the CPU utilization by file size as it shows the same dip in the 90 CPU boundaries. This chart may help to strengthen the what is seen in Fig 9.

VI. Conclusions and Future Work

It is clear that the NX Parser is the most efficient and may be the best for applications needing no validation. The Any23 parser is comparable to the NX Parser in performance but not in efficiency.

In order to understand why Jena is the slowest, the Jena parser has a larger codebase and is performing validations while parsing. Any23 and the NX Parser are doing simple format validations before writing to the destination output.

Computing speed, dataset size and quality of output are the critical parameters to be considered when determining an application's needs. Usually we want processing to be instantaneous regardless of the dataset size but this is not realistic. If the application has extremely large datasets and quality is critical, then we will have to give up speed and use the Jena processor. For some extremely large datasets, Jena may not be feasible. On the other hand, if the application with extremely large datasets is not subject to quality requirements or data validation has already been performed on the dataset, e.g. the Web Data Commons files, then the NX Parser or Any23 are recommended. A pre-validation step using a specialized format checker may be required for some applications that require both quality and performance when using the NX Parser or Any23.

It is also clear that there is acceptable speedup when using the Amazon EMR with the smaller file sizes, i.e. 0.7 and 4.7 GB. At the large 15.9 GB file size, the performance seemed to level off and the results were similar or worse than the smaller files. However, it is not clear where the best or optimal cost to performance curve might be. That is up to more testing of the RDFParseParallelMain program which could be done in some future work. But it is interesting to witness the various performance curves within the scenarios of file size and CPU count. The charts can be used for each parser as a starting point when the file size is known. This could prevent overspending of CPU usage for any given test.

Hopefully, this work has provided a rich set of information about tackling Big Data and Semantic Web challenges. More specifically, when your challenge is to query large RDF datasets, the determination of which parser to use may now be much easier. This work has been an exhausting challenge. But with any activity, there is always more to do. Next we will highlight some of the areas where future investigation could be focused.

This work did not have the time or budget to perform extensive testing. Even though the testing program and methodology are very effective and produced reasonable results, more testing could solidify the observed trends. Larger dataset sizes would put higher stress on the parsers and CPU's. Using more and faster CPU's could result in a better understanding of the speedup curve. CPU analysis could also indicate where there is a diminishing return. In other words, the use of different CPU models could give the optimal approach for running certain application types.

Even though this work was performed on pre-validated datasets, un-validated datasets could be tested and, along with the above observations, a better heuristic could be developed for the determination of use for a variety of applications. In other words, there may be combinations of parsers and validators that could run in a sequence of steps optimized for a particular application. And, there are many CPU combinations that could be considered for the best result. For example, a very large dataset requiring an average level of validation might optimize at 200 large standard, first-generation instances. A chart could be developed based on the application or scenario type, as well.

References

[1] Retrieved October 2012, from Web Data Commons: http://webdatacommons.org/.

[2] Retrieved October 2012, from Common Crawl: http://commoncrawl.org/.

[3] Retrieved February 2013, from Amazon EC2: https://aws.amazon.com/ec2/.

[4] Retrieved February 2013, from Amazon EMR: http://aws.amazon.com/elasticmapreduce/.

[5] Retrieved October 2012, from Apache Any23: http://any23.apache.org/.

[6] Retrieved 02 2013, from Apache Jena: http://jena.apache.org/index.html

[7] Retrieved October 2012, from NX Parser: https://code.google.com/p/nxparser/ redirected from http://sw.deri.org/2006/08/nxparser/.

[8] Retrieved April 2013, from WikiPedia: Big Data: http://en.wikipedia.org/wiki/Big_data.

[9] Retrieved February 2013, from Amazon S3: http://aws.amazon.com/s3/.

[10] Retrieved October 2012, from RDF NQuads: http://sw.deri.org/2008/07/n-quads/.

[11] Retrieved November 2013, from Apache Jena Tutorial: http://jena.apache.org/tutorials/rdf_api.html.

[12] Retrieved October 2013, from Any23 Incubator: http://incubator.apache.org/projects/index.html#graduated.

[13] Retrieved October 2012, from NxParser Deri: http://semanticweb.org/wiki/DERI_Galway.

[14] Retrieved October 2012, from Nxparser Yars.

[15] Mohammad Farhan Husain, e. a. (2010). Data Intensive Query Processing for Large RDF Graphs Using Cloud Computing. *IEEE.*

[16] Raffael Stein, e. a. (2009). RDF on Cloud Number Nine.

[17] Bugiotti, F. (2012). RDF Data Management in the Amazon Cloud. *DanaC 2012.*

[18] Hannes Mühleisen, e. a. (2012). Web Data Commons – Extracting Structured Data from Two Large Web Corpora. *IEEE.*

[19] Ho, R. (2008, 12 16). *How Hadoop Map/Reduce works*. Retrieved April 2013, from DZone: http://architects.dzone.com/articles/how-hadoop-mapreduce-work.

[20] Retrieved January 2013, from Apache Hadoop: http://hadoop.apache.org/.