Abstract

This project examines the usage of the Hadoop framework for textual analysis. Two disparate datasets are analyzed through the course of the project on an extremely hardware-heterogenous Hadoop cluster. Results of the textual analysis are discussed as well as varying optimizations that were implemented to ensure peak performance of all systems and utilized algorithms.

Introduction

Over 90% of the world's data was generated within the last two years [1]. Following an exponential trend that has exceeded even Moore's famous law for processors [2], our ability to store data per-capita has reached levels that were largely inconceivable just a decade ago. This spectacular increase in the amount of data has essentially broken many of the traditional systems and algorithms that are relied upon to manage the data. The emergence of the "Big Data" field was an effort to find solutions to the ever-growing challenges of working effectively with data sets that are simultaneously growing more massive and more important to our daily lives. Though there are many different projects under heavy development, the Hadoop framework occupies center stage in our present Big Data toolset.

The goal of this project was to focus on a particular segment of this growing avalanche of data: natural text. The explosive rise of social media over the past decade combined with the increased technological resources at the disposal of free "text aggregation" services such as Project Gutenberg has resulted in a proliferation of text-based opinions, knowledge, and nonsense. Gaining insight from such massive and often disorganized data poses serious challenges but is a task well-suited for a framework such as Hadoop.

Throughout the course of handling the data sets (setup instructions described in-depth in Appendix B), several Hadoop "ecosystem" projects were a particularly good fit to assist in solving aspects of the problem. These projects include Hive, Pig, Oozie, Sqoop, and most prominently, Mahout.

Problem Description and Specification

Two different data-sets were utilized during the course of the project: hundreds of millions of sentences in various languages from the Leipzig Corpora collection and tens of thousands of full-text books from Project Gutenberg. One attractive feature of the Leipzig Corpora is the availability of previously-computed analysis done on subsets of the data that make it possible to compare initial results to those computed through conventional means before moving on to larger portions of the data set. In addition to this analysis (easily accessible via the CorpusBrowser interface discussed in Appendix C), individual partitions of the massive data-set can be downloaded as traditional SQL databases and interacted with through familiar RDBMS syntax [3].

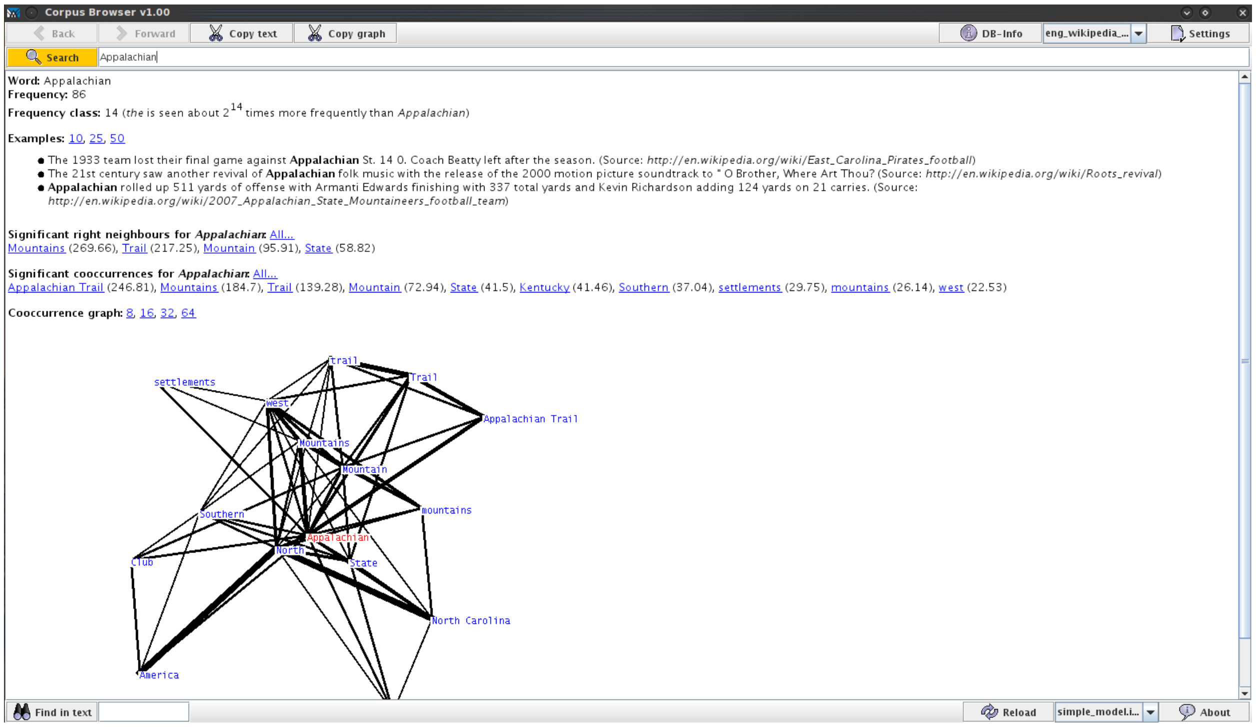
The overall size of the Leipzig Corpora data (approximately 240 GB) is vastly out of any of our cluster machines' ability to store in-memory. Working with such data and the heavy hard disk I/O times it implies means that Hadoop often takes considerable time to perform operations. What it may sacrifice in speed however it attempts to make up for in scalability and power. With Hadoop it is possible to aggregate the disparate conventional databases that comprise the Corpus and perform queries and analytics on all of them at once.

In addition to being able to query and interact with the Leipzig data on a much broader scale than conventional methods would allow for, another goal of this project is to learn more about the data contained within the Corpus. Conventional programs for clustering, classification, and machine learning would have great difficulty dealing with the sheer size of the Corpus, taking a prohibitively long time to return answers if they were able to do so at all. Hadoop and its Mahout machine learning library are utilized to fill these holes and gain insight into the textual data that would have been extremely difficult or impossible to accomplish previously. Being able to successfully cluster even small sentences according to their native languages is a challenging prospect but one that Mahout proved capable of. When the resultant clusters were fed into a classification system, the program was able to correctly classify over 99% of newly inputted sentences.

For all the attractive qualities of the Leipzig Corpora, the text data dealt with is but sentences pulled at random from various sources. It would therefore be difficult for any algorithm or possibly even any human to accurately group such sentences by their original topics. Project Gutenberg, by comparison, offers documents of vast length and great potential for thorough topic analysis. Though the project offers tens of thousands of completely free ebooks, the books themselves remain entirely unorganized and difficult to search for an interested party seeking a book on a certain topic. By working extensively with the collection and applying a stochastic Collapsed Variational Bayesian algorithm for Latent Dirichlet Allocation, it is shown that the collection can in fact be sorted successfully by topic and genre.

Problem Solution : Leipzig Corpora - Basic Analysis

As previously mentioned, the Leipzig Corpora CorpusBrowser application makes it simple to browse previously-computed statistics pertaining to an individual corpus. In the figure below, the CorpusBrowser displays information specific to the word “appalachian” in the English Wikipedia Corpus of 2010.



Word: Appalachian
Frequency: 86
Frequency class: 14 (*the* is seen about 2^{14} times more frequently than *Appalachian*)

Examples: 10, 25, 50

- The 1933 team lost their final game against **Appalachian** St. 14 0. Coach Beatty left after the season. (Source: *http://en.wikipedia.org/wiki/East_Carolina_Pirates_football*)
- The 21st century saw another revival of **Appalachian** folk music with the release of the 2000 motion picture soundtrack to " O Brother, Where Art Thou? (Source: *http://en.wikipedia.org/wiki/Roots_revival*)
- **Appalachian** rolled up 511 yards of offense with Armanti Edwards finishing with 337 total yards and Kevin Richardson adding 124 yards on 21 carries. (Source: *http://en.wikipedia.org/wiki/2007_Appalachian_State_Mountaineers_football_team*)

Significant right neighbours for *Appalachian*: All...
Mountains (269.66), Trail (217.25), Mountain (95.91), State (58.82)

Significant cooccurrences for *Appalachian*: All...
Appalachian Trail (246.81), Mountains (184.7), Trail (139.28), Mountain (72.94), State (41.5), Kentucky (41.46), Southern (37.04), settlements (29.75), mountains (26.14), west (22.53)

Cooccurrence graph: 8, 16, 32, 64

***Figure 1**: CorpusBrowser visualization of "Appalachian"*

Through the above visualization, we see that the word "Appalachian" was encountered 86 times in the corpus. Common bigrams include "Appalachian Mountains," "Appalachian Trail," "Appalachian Mountain," and "Appalachian State."

To begin performing similar analysis of the entire corpora in Hadoop, the AutoSqoop project was created and utilized to automate the importation of the numerous Leipzig component databases to HDFS (see Appendix C for usage instructions).

Once the data is resident on HDFS, the Hadoop framework offers a variety of paradigms through which to interact with the data. In order to determine the most efficient way to manage such data, the same analysis performed above for the work "Appalachian" (word count, left/right neighbor count, sentence co-occurrence) was performed through MapReduce, PigLatin, HiveQL, PigLatin with optimized User Defined Functions (UDFs), and HiveQL with UDFs. To automate the deployment of these various processing tests, a directed acyclic graph (DAG) workflow was developed with Apache Oozie, allowing for one-line deployment and automation of the tests.

Prior to Hadoop 2 and the introduction of YARN, MapReduce was the only processing model one could employ on Hadoop data. While the framework has now been expanded to include other processing paradigms, the MapReduce algorithm still remains the quintessential method for interacting with Hadoop data and is thus the first logical choice to perform analysis on the data. The Map() function employed during the Map phase of this process is shown in Figure 2.

```
public static class CountsMapper extends Mapper<Object, Text, Text, IntWritable>
{
        private final static IntWritable ONE = new IntWritable(1);
        private Text word = new Text();

        public void map(Object key, Text value, Context context) throws
IOException, InterruptedException
        {
                Configuration conf = context.getConfiguration();
                String wordToSearch = conf.get("wordToSearch").toLowerCase();
                word.set(wordToSearch);
                String[] words = value.toString().toLowerCase().split("[,. \"]");
                for (int i = 0; i < words.length; i++)
                        if (words[i].compareTo(wordToSearch) == 0)
                                        context.write(word, ONE);
        }
}
```

***Figure 2**: Wordcount Map() implementation*

When given an input folder with the files containing the millions of Corpora sentences, the Hadoop framework will feed them to the Map() function feature in FIgure 2. If the data was imported using the AutoSqoop framework, the initial key will be a LongWritable value indicating the byte offset in the file that the current Text (sentence) value was found at. If the files were stored in SequenceFile format, the Key will be a Text value containing the name of the origin database and an index therein. Because of the various formats possible and because the Mapper key in this instance isn't actually used for anything, the argument was abstracted to be of Object type.

The Text class seen above is simply a wrapper class for Java's String; likewise IntWritable is simply a wrapper for a Java int. Similar classes (BooleanWritable, ByteWritable, etc.) exist for all Java primitives and many common objects such as String. The impetus behind their development was the limitations of Java's default Serialization library. While effective for most use cases, the default framework fails in distributed environments. One reason is that Serializable Java classes by default write their class names and object representations to the data stream each time that a first instance of that class is detected (subsequent encodings contain a pointer to the original representation) in a stream. While this is convenient for a solitary receiver, when the data is going various places one or more recipients are likely to end up without a class representation to work off of. A solution therefore requires that representations be written for all instances of serialized objects, or none at all. Hadoop's Writable format employs the second strategy. In order to eliminate the problems with Java's serialization format as well as to cut down on the size of serialized data being passed around the cluster through RPC, the receivers are always expected to know which data type they're expecting and its interface. Once received, the deserialized objects can be treated just like their Java equivalents [4].

In the Mapper featured in FIgure 2, the Text sentence value is passed in, cast to lower case, and split on spaces. If the wordToSearch ("appalachian" in this instance) is encountered, a key-value pair consisting of ("appalachian": 1) is output. The subsequently run Reducer (Figure 3) is responsible for aggregating for aggregating the encountered instances. If so many instances of the word were encountered that more than one reduce()

function is required, the various reducers will output their respective sums to another generation of reducers which will sum the prior sums.

```
	public static class CountsReducer extends Reducer<Text, IntWritable, Text, 
IntWritable>
	{
		public void reduce(Text key, Iterable<IntWritable> values, Context context) 
throws IOException, InterruptedException
		{
			int sum = 0;
			for (IntWritable num : values)
				sum += num.get();
			context.write(key, new IntWritable(sum));
		}
	}
```

Figure 3*: Wordcount Reduce() implementation*

Similar MapReduce solutions for sentence co-occurences and left and right neighbor counts were implemented. Table 1 displays the results of the various MapReduce processes run against a standardized subset of the corpora.

	Word Search	**Left Neighbor**	**Right Neighbor**	**Co-occurrence**
MapReduce	29.6	41.6	38	37.2

Table 1*: MapReduce analysis time results*

While MapReduce is exceptionally powerful and fast - a description well supported by the comparative analysis to come - it can be tedious to write MapReduce map() and reduce() functions for all cases, especially when those tasks are not likely to need repeating.

Pig

Using Pig and its constituent language, PigLatin, many common MapReduce functions can be implemented quickly and in but a few lines of syntax. PigLatin provides a rich library of complex data structures and transformative functions that may be applied to specified data. Internally, the transformations are turned into a series of MapReduce jobs, launched automatically by the framework [5]. Figure 4 features an example PigLatin program to calculate the right neighbors of the word “appalachian.”

```
A = load '$input';
B = foreach A generate LOWER(REGEX_EXTRACT($0, '(?<= [aA]ppalachian )(\\w+)', 1)) as 
word;
C = FILTER B BY word MATCHES '\\w+';
D = GROUP C By word;
E = foreach D generate COUNT(C), group;
store E into '$output/pigRight';
```

Figure 4*: PigLatin implementation of right neighbor search*

Clearly, PigLatin provides a much easier route for simple, common tasks. Of additional note is the presence of a Java-style regular expression, used in this instance to extract the right neighbor. Since PigLatin - like nearly all Hadoop ecosystem projects - is written in Java, the aforementioned regular expression does in fact depend on the Java regular expression library.

An additional bonus of being written in Java is the ease of extending PigLatin’s capabilities through the creation of User-Defined Functions (UDF). While the code shown in Figure 4 is certainly more simple and easy to write than the prior MapReduce example, having to use regular expressions to extract the right neighbors is still a bit convoluted and possibly performance-taxing. Figure 5 demonstrates a PigLatin UDF implementation for Right Neighbor analysis, and Figure 6 shows how such a UDF can be utilized in subsequent executions to increase readability and performance.

```
public class RightNeighbors extends EvalFunc<String>
{
       @Override
       public String exec(Tuple tuple) throws IOException
       {
              if (tuple == null || tuple.size() == 0)
                     return null;
              String line = ((String)tuple.get(0)).toLowerCase();
              String word = (String) tuple.get(1);
              // STDOUT, STDERR -> HDFS logDir
              if (line.contains(word))
              {
                     String[] words = line.split(" ");
                     for (int i = 0; i < words.length; i++)
                            if (words[i].equals(word))
                                   if ((i+1) < words.length)
                                          return words[i+1];
              }
              return null;
       }

       @Override
       public List<FuncSpec> getArgToFuncMapping() throws FrontendException
       {
              List<FuncSpec> funcList = new ArrayList<FuncSpec>();
              Schema scheme = new Schema();
              scheme.add(new Schema.FieldSchema(null, DataType.CHARARRAY));
              scheme.add(new Schema.FieldSchema(null, DataType.CHARARRAY));
              funcList.add(new FuncSpec(this.getClass().getName(), scheme));
              return funcList;
       }
}
```

Figure 5: *PigLatin UDF RightNeighbor implementation*

```
REGISTER '$lib/PigCounts-1.0.jar';
DEFINE rightNeighbors edu.appstate.kepplemr.pigcounts.RightNeighbors();
A = load '$input' AS (sent:chararray);
B = FOREACH A GENERATE rightNeighbors(sent,'appalachian') AS word;
C = group B by word;
D = foreach C generate COUNT(B), group;
store D into '$output/pigUDFRight';
```

Figure 6: *PigLatin UDF execution*

UDF implementations will often be faster than their built-in alternatives due simply to the more streamlined logic that can be employed when the code author knows exactly what they want and how to get it. Another optimization that can be employed eliminates the need to load the data and then iterate through it. By defining a custom data loading UDF, only data adhering to a certain schema will be projected and loaded into variable ‘A’. Figure 7 displays such a UDF implementation in Java.

```
public class CountWord extends LoadFunc
{
        ...
        @Override
        public Tuple getNext() throws IOException
        {
                try
                {
                        if (!reader.nextKeyValue())
                                return null;
                        String line = ((Text)
reader.getCurrentValue()).toString().toLowerCase();
                        Tuple tuple = tupleFactory.newTuple(1);
                        int index = line.indexOf(word);
                        int count = 0;
                        while (index != -1)
                        {
                            count++;
                            line = line.substring(index + 1);
                            index = line.indexOf(word);
                        }
                        tuple.set(0, count);
                        return tuple;
                }
                catch (InterruptedException e)
                {
                        throw new ExecException(e);
                }
        }
}
```

***Figure 7**: Load UDF implementation for word count*

Now that so much of the work is done during the initial load, the calling PigLatin program has much less to do, as shown in Figure 8.

```
REGISTER '$lib/PigCounts-1.0.jar';
A = load '$input' USING edu.appstate.kepplemr.pigcounts.CountWord(' appalachian ') AS
(num:int);
B = GROUP A BY num;
C = FOREACH B GENERATE SUM(A.num);
store C into '$output/pigUDFCount';
```

***Figure 8**: PigLatin program calling Load UDF*

Table 2 displays the results of the various counting implementations run in base PigLatin and through PigLatin with UDF implementations.

	Word Search	Left Neighbor	Right Neighbor	Co-occurrence
Pig	55.6	65.6	60.6	49.4
Pig UDF	49.6	50.4	49	58.4

Table 2*: PigLatin analysis time results*

Hive

Last on the list of Hadoop processing frameworks utilized by this project is Hive, Hadoop's data warehousing and querying tool. Apache Hive allows users to aggregate vast amount of data according to defined schemas. Once the data has been aggregated the HiveQL querying language can be used to interact with it. HiveQL was heavily influenced during its development by MySQL [6], and it attempts to allow users not entirely familiar with the Hadoop stack to interact and learn from data in ways that are more familiar to them. As an example of its power, the following query is all that is needed to combine all the various AutoSqoop-output databases into one easily-queryable collection:

```
DROP TABLE corpora;
CREATE EXTERNAL TABLE corpora
(
  sentence STRING
)
row format delimited fields terminated by '\n'
location '${table}';
```

Figure 9*: HiveQL Schema table creation for flat-text files.*

Or, if working with SQL2Seq SequenceFile output formats:

```
CREATE EXTERNAL TABLE IF NOT EXISTS sequencefile_table
(
       sentence STRING
) stored as sequencefile;
LOAD DATA INPATH "/user/hadoop/seqout" INTO TABLE sequencefile_table;
```

Figure 10*: HiveQL Schema table creation for SequenceFile files.*

Once the corpora table has been created and loaded with all sentences in the corpora, a specific wordcount can be done in the following way:

```
DROP TABLE IF EXISTS wc;
CREATE TABLE wc AS SELECT word, count(1) AS count
FROM
(
  SELECT EXPLODE(SPLIT(LOWER(sent), ' '))
  AS word
  FROM
   (
    SELECT sentence AS sent
    FROM corpora
    WHERE INSTR(LOWER(REGEXP_REPLACE(sentence,'[\\p{Punct},\\p{Cntrl}]','')),
"appalachian") > 0
  ) xxx
) w GROUP BY word ORDER BY count DESC, word ASC;
```

```
INSERT OVERWRITE DIRECTORY '${output}/hiveCount'
SELECT word,count FROM wc
WHERE INSTR(LOWER(word), "appalachian") != 0;
```

Figure 11: *HiveQL query for "appalachian" word count*

The resultant wc table will also contain information regarding sentence co-occurrence. To generate counts of left and right neighbors, a different - and more complicated - approach is needed. Figure 12 demonstrates a HiveQL query for calculating and counting left neighbors.

```
DROP TABLE IF EXISTS wc;
CREATE TABLE wc AS SELECT word, count(1) AS count
FROM
(
  SELECT REVERSE(SPLIT(LTRIM(REVERSE(LOWER(SUBSTR(sent, 0, INSTR(LOWER(sent),
"appalachian")-2)))), ' ')[0])
  AS word
  FROM
  (
    SELECT sentence AS sent
    FROM corpora
    WHERE INSTR(LOWER(sentence), "appalachian") > 0
  ) x
) w GROUP BY word ORDER BY count DESC, word ASC;
INSERT OVERWRITE DIRECTORY '${output}/hiveCountLeft'
SELECT word,count FROM wc;
```

Figure 12: *HiveQL query for left neighbors*

Fortunately, HiveQL also supports the creation of UDFs to simplify (and speed-up) query execution.

```
public class LeftNeighbors extends UDF
{
      private Text result = new Text();
      public Text evaluate(Text line, Text word)
      {
            String sentence = line.toString().toLowerCase();
            String token = word.toString().toLowerCase();
            if (sentence.contains(token))
            {
                  String[] words = sentence.split(" ");
                  for (int i = 0; i < words.length; i++)
                        if (words[i].equals(token) && (i > 0))
                        {
                              result.set(words[i-1]);
                              return result;
                        }
            }
            return null;
      }
}
```

Figure 13: *HiveQL Left Neighbor UDF implemenation*

After incorporating the UDF, what once was a rather complicated query is now rendered much more palatable (Figure 14).

```
ADD JAR ${lib}/HiveCounts-1.0.jar;
CREATE TEMPORARY FUNCTION left AS 'edu.appstate.kepplemr.hivecounts.LeftNeighbors';
```

```
DROP TABLE IF EXISTS wc;
CREATE TABLE wc AS SELECT word, count(1) AS count
FROM
(
    SELECT left(sentence, 'Appalachian') as word
       FROM corpora
       WHERE left(sentence, 'Appalachian') IS NOT NULL
) w
GROUP BY word ORDER BY count DESC, word ASC;
INSERT OVERWRITE DIRECTORY '${output}/hiveCountUDFLeft'
SELECT word,count FROM wc;
```

Figure 14*: HiveQL Left neighbor query with UDF*

Table 3 displays the results of the various counting implementations run in base HiveQL and through HiveQL with UDF implementations. While the performance of most of the UDF's closely mirrors that of their conventional counterparts, a significant performance increase is seen in the basic word count implementation. This is due, similar to the PigLatin load function example, to processing the data in memory only once - upon initial loading each sentence maps only to the number of times the key word - "Appalachian" - was encountered on the first pass through. Once these results are tabulated, no grouping is required because it is known that only one word was being search for.

	Word Search	**Left Neighbor**	**Right Neighbor**	**Co-occurrence**
Hive	95.8	94.6	97.2	100.2
Hive UDF	42	100.2	97.8	96.2

Table 3*: HiveQL analysis time results*

As another example of the power of HiveQL for data analysis, when it was used to examine the occurrence of "Appalachian" across the years covered in the datasets (Figure 15), a noticeable spike was found to occur in the year 2007. Accompanying this sudden rise in frequency was a large jump in the co-occurrence frequency of terms "Armanti", "Michigan", "Subdivision", and "I-AA", words strongly associated with the Appalachian State football team's surprise victory over the University of Michigan football team on September 1st, 2007.

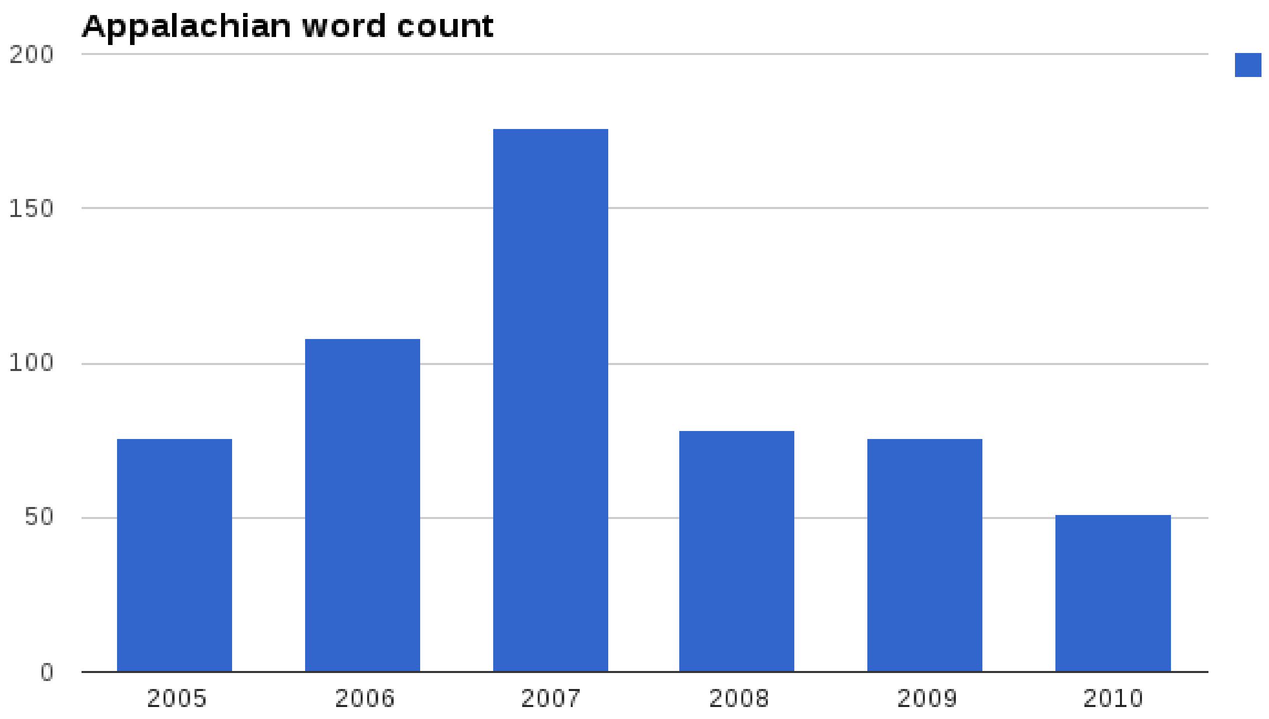


***Figure 15**: Appalachian word occurrence in news documents*

The overall analysis of the cluster resulted in - almost - identical results through all of the different paradigms employed. Tables 4-7 show the results for the “Appalachian” word analysis over the entire Corpora dataset.

Eng_Wiki_2010	86
Corpora	1905

***Table 4**: Corpora-wide word search results*

the	2686
of	1132
in	1028
and	992
and	860
to	830
state	541
for	378
on	284
is	272

***Table 5**: Sentence co-occurrences*

state	469
trail	179
mountains	135
power	62
league	46
mountains	40
state's	38
regional	32
and	28
spring	16

***Table 6**: Right neighbors*

the	520
NULL	244
at	76
and	52
of	52
southern	50
to	44
a	43
against	30
an	28

***Table 7**: Left neighbors*

With regards to efficiency, the varying methods were timed and compared in their analysis of the corpora data:

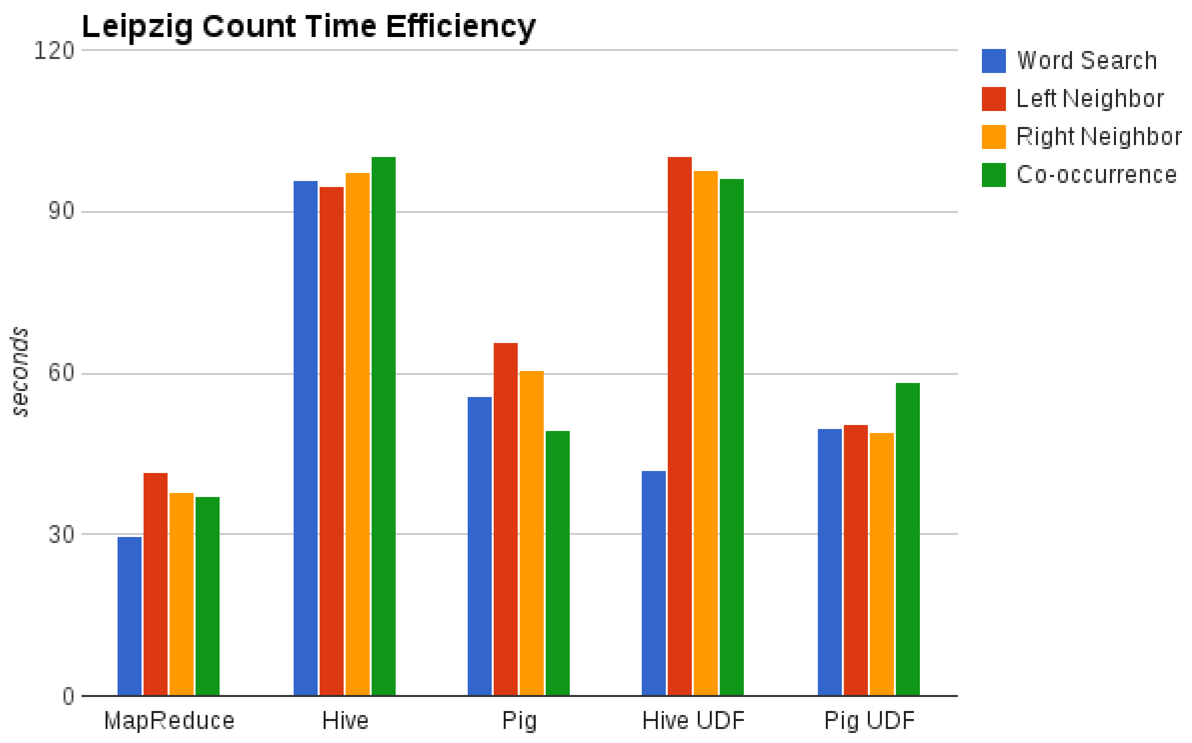


Figure 16: *Cross-platform search efficiency benchmarks*

Problem Solution : Leipzig Corpora - Clustering by Language

Clustering the Leipzig data according to language provided the first opportunity to integrate the Leipzig data with the Mahout framework. In order to do so the data had to be converted from the format produced by Sqoop into a <Text, Text> format accessible to Mahout. To automate this process the SQL2Seq project was created and utilized.

In order to easily recognize the various clusters only 8 languages were used for clustering (English, Spanish, German, French, Indian, Russian, Dutch, and Japanese), accounting for roughly 160 GB of the total size of the Corpora.

The clustering algorithm utilized was K-Means run through Java calls to the Mahout library. The corpora sentences were first broken down into their constituent words, with each sentence represented as a vector in the vector space model with a dimension for each occurring word in the corpora. Any words present in the sentence are assigned a value according to the term frequency-inverse document frequency model defined by the equations displayed in equations 1 and 2.

$$\mathrm{tf}(t,d) = 0.5 + \frac{0.5 \times \mathrm{f}(t,d)}{\max\{\mathrm{f}(w,d) : w \in d\}} \qquad (1)$$

$$\mathrm{idf}(t) = \log \frac{|D|}{1 + |\{d : t \in d\}|} \qquad (2)$$

Where:

t -> term.
d -> document

max{ f(w,d) : w ∊ d} -> frequency of the most frequently occurring term in the document; used to compute relative frequency and prevent bias toward longer documents.
|*D*| -> number of documents in the corpus.
(1 + |{d : t ∊ d } |) -> number of documents where the term ‘t’ appears.

The basic idea behind running KMeans through a distributed MapReduce paradigm is simple: the Map phase reads the stored cluster centers and iterates over all sentence vectors, calculating the closest available distance to a cluster center by some defined distance metric. At the end of the Map() phase the computed clusters are stored with their associated vectors [7].

The Reduce() phase iterates through all members of a cluster determined in the Map() phase and computes a new centroid based upon the average distance between all of its members. The new center is then compared against the old one to check if suitable convergence has been achieved].

The dimensionality of nearly any text data is likely to be extremely large, and even more so when words are present from a variety of languages. The dictionary size when utilizing the Corpora from just the eight languages chosen exceeds 40 million entries.

The purpose of the KMeans algorithm is to cluster similar items together. To accomplish this task, the algorithm must be able to accurately calculate the similarity and dissimilarity of items. The equations that capture such metrics are known as distance measures. The default distance measure utilized by the Mahout framework is the Euclidean distance measure, with its equation shown in equation 3.

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + ... + (p_i - q_i)^2 + ... + (p_n - q_n)^2}. \qquad (3)$$

The Euclidean distance measure is an extremely effective and easy solution for many problems which mimics the way most humans think of distance. It tends to suffer however from the “curse of dimensionality” and its performance declines rapidly as data becomes as vast and dimensionally-laden as the Leipzig sentences. When so many dimensions are present, all space tends to be far away from the center yet very close to one another, owing to the sparse nature of the origin vectors and how infrequently their non-zero values will intersect. Eventually the distance to the nearest data point approaches the distance to the farthest data point [8].

A distance measure capable of better dealing with high-dimensional data was needed, and the two best measures utilized in the project were the Cosine similarity measure and the Tanimoto distance measure (also known as the Jaccard similarity coefficient). Though dimensionality effects these measures to some extent as well, generally speaking they perform better than Euclidean measures on high-dimensional data, and text data in particular [9].

The Cosine similarity measure (equation 4) utilizes the cosine of the angle formed in the distance evaluations of two vectors. Each dimension represents a term with its TF-IDF weight in the document, which must be non-negative. The cosine measure by its nature is naturally normalized, meaning that recursively concatenating a document to itself will not alter its similarity measure with the cosine measure, though it would in Euclidean measures and others. As long as the composition of documents is identical, the specific document totals do not matter. In addition, the cosine measure tends to be quick to evaluate, as only the non-zero dimensions are considered. Given that it is the angle that matters in this measure, it is also more resistant to the zero-convergence of distance seen in the Euclidean measure when dealing with high-dimensional data.

The Tanimoto distance measure is detailed in equation 5. For textual data, the algorithm compares the total weight of shared words to the weight of words featured in one sentence but not the other. Like the cosine measure, the Tanimoto measure ranges between 0 and 1; 1 when all terms are the same and 0 when all terms are disjoint.

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}} \tag{4}$$

$$d = 1 - \frac{(a_1 b_1 + a_2 b_2 + \dots + a_n b_n)}{\sqrt{(a_1^2 + a_2^2 + \dots + a_n^2)} + \sqrt{(b_1^2 + b_2^2 + \dots + b_n^2)} - (a_1 b_1 + a_2 b_2 + \dots + a_n b_n)} \tag{5}$$

The project KMeans implementation utilizes features of the Mahout 1.0 pre-alpha libraries that are bundled along with it. Using the 1.0 libraries was necessary to get the code running on the Hadoop 2 (YARN) cluster employed throughout the analysis. The KMeans implementation interface is displayed in Figure 17.

```
Usage:
 [--k-value <<num_topics>> --input <<input>> --output <<outputDir>>
--maxDFpercent <<maxPercent>> --minDFpercent <<minPercent>> --skipSeqDir
--skipSparse --skipClustering --maxIter <<# iterations>>]
Options
  --k-value (-k) <num_topics>              number of topics to search for
  --input (-i) <input>                     text document directory to cluster
  --output (-o) <outputDir>                output directory for the various
                                           stages
  --maxDFpercent (-maxDf) <maxPercent>     maximum document frequency allowed
                                           for terms
  --minDFpercent (-minDf) <minPercent>     minimum document frequency allowed
                                           for terms
  --skipSeqDir (-s1)                       skip SequenceFile generation stage
  --skipSparse (-s2)                       skip SparseVector generation stage
  --skipClustering (-s3)                   skip KMeans clustering stage
  --maxIter (-m) <# iterations>            maximum clustering iterations to
                                           perform
```

***Figure 17:** TextProject KMeans interface.*

When the KMeans run was completed, the ClusterDump utility can be utilized to view the resultant clusters and their most significant word associations, with those shown in Figure 18 corresponding to English and Spanish clusters:

```
mahout clusterdump -dt sequencefile -d
hdfs://aho:8020/user/hadoop/Lang_Out/sparseVectors/dictionary.file-0 --input
hdfs://aho:8020/user/hadoop/Lang_Out/kmeans/clusters-30-final --output
/home/hadoop/lang8.txt -b 10 -n 10
```

```
...
:VL-129942{
        Top Terms:
                he                                      =>  0.4321995088141447
                his                                     => 0.35200716884713684
                said                                    =>  0.3464244438552207
                from                                    =>  0.3382853389657676
                have                                    =>  0.2981334591790388
                has                                     =>  0.2903694849275929
                i                                       =>  0.2664428856072751
                were                                    => 0.21967396370770398
                had                                     =>  0.2193561644182676
                who                                     => 0.21803578885692346
:VL-129943{
        Top Terms:
                de                                      =>   1.935401108220715
                la                                      =>    1.41327124929813
                le                                      =>  1.2486838799570223
                et                                      =>  1.1165280979879373
                à                                       =>  1.0691059800650915
                les                                     =>  1.0340855076165913
                des                                     =>   0.926662732820834
                en                                      =>  0.9255334632971403
                du                                      =>  0.7806759012191344
                un                                      =>  0.7049445195880997
...
```

***Figure 18**: K-Means clustering by language results*

Problem Solution : Project Gutenberg - Topic modelling through LDA

In the analysis of the Leipzig Corpora performed to cluster sentences based upon language, vectors were generated for each sentence and sentences were clustered according to similarities between them. By comparison, the stochastic Collapsed Variational Bayesian algorithm employed on the Project Gutenberg etexts is a posterior inference algorithm for determining topics generated through a Latent Dirichlet Allocation (LDA) generative process. It attempts to “reverse” the generative LDA process and stochastically determine to the topics of documents assumed to have been created through an LDA process [10].

In the generative process of Dirichlet Allocation, the document is assumed to have chosen k-number of topics to be composed of. For each document, a distribution of the topics is chosen and words associated with the topics are distributed according to a distribution related to the specific document. Though this algorithm is of course not what actually created the texts being analyzed, the posterior inference methods utilized subsequently in order to guess what the k-topics originally were also do a good job on documents not actually created in such a manner.

Prior implementations of LDA used traditional inference techniques such as Gibbs sampling and standard variational inference. These methods proved effective, but required an enormous amount of computation for any reasonably-sized corpora. These algorithms are more compute-intensive because they assign random values to not only the latent variables desired but also the parameters of the model. The “collapsed” implementation used in the current version of Mahout marginalizes the parameters out and focuses only upon the latent variables. It has been shown that it is possible to perform inference in this collapsed representation and then subsequently go back and recover estimates of the parameters [11].

The “variational” qualifier in CVB results from its employment of variational inference. The purpose of these methods is to transform the problem in question into an optimization problem. This is done through the use of variational parameters, which when individually fixed can often make the problem solution much easier to calculate. The goal is to find a probability density as close as possible to the true posterior, and the variational parameters are approximations using known distributions, attempting to minimize Kullback–Leibler (KL) divergence.

In order to further increase the speed of the algorithm when operating on large corpora, the set of input documents is also now split into batches so that the algorithm can iterate on a smaller subset and draw observations before moving on to the entire corpora. A greatly simplified pseudocode approximation of the CVB algorithm is featured in Figure 19.

```
Randomly initialize wordTopics, documentTopics;
Set overallTopics = current corpusTopics;
For each batch of documents:
    For each document:
        for each “burn-in” phase:
            For each token:
                update variational distribution for word (6)
```

```
                    update documentTopics; (7)
          For each token in document
               update variational distribution for word
               update documentTopics;
               re-estimate topic for word in document.
               re-estimate topics in batch
         update wordTopics; (8)
         update overallTopics; (9)
```

Figure 19*: CVB pseudocode*

$$\gamma_{ijk} :\propto \frac{N^{\Phi}_{w_{ij}k} + \eta_{w_{ij}}}{N^{Z}_{k} + \sum_{w} \eta_{w}} (N^{\Theta}_{jk} + \alpha) \ . \quad (6)$$

$$\mathbf{N}^{\Theta}_{j} := (1 - \rho^{\Theta}_{t})\mathbf{N}^{\Theta}_{j} + \rho^{\Theta}_{t} C_{j} \gamma_{ij} \ . \quad (7)$$

$$\mathbf{N}^{\Phi} := (1 - \rho^{\Phi}_{t})\mathbf{N}^{\Phi} + \rho^{\Phi}_{t} \hat{\mathbf{N}}^{\Phi} \quad (8)$$

$$\mathbf{N}^{Z} := (1 - \rho^{\Phi}_{t})\mathbf{N}^{Z} + \rho^{\Phi}_{t} \hat{\mathbf{N}}^{Z} \quad (9)$$

Where:
C^{J} -> Number of words in document j.
w_{ij} -> Dictionary index for word (i,j)
α -> Dirichlet prior parameters for θ
η -> Dirichlet prior parameters for η
γ_{ij} -> Variational distribution for word (i, j)
N^{Θ} -> Expected topics counts per document
N^{ϕ} -> Expected topic counts per word
N^{Z} -> Expected topic counts overall
$\hat{N}^{\Theta}$ -> Estimate of N^{Φ} from current batch
$\hat{N}^{Z}$ -> Estimate of N^{Z} from current batch
ρ_{t}^{Θ} -> Step size for N^{Θ} at time t
ρ_{t}^{ϕ} -> Step size for N^{Φ} and N^{Z} at time t

Though the new CVB algorithm offers substantial speedup over previous versions of the LDA algorithm, analyzing a text such as the Gutenberg project is still a significant undertaking. In its original form, the Gutenberg text contains over 80,000 full text documents consisting of over 35 million unique tokens. To trim this dataset before running it through the CVB algorithm - necessary to fit into the available heap space of many cluster machines - a new Lucene Analyzer was implemented for the Gutenberg corpora, the GutenbergAnalyzer. The GutenbergAnalyzer wraps several previously-created Lucene filters through a decorator pattern for filtering length, “stop” words, and conducting Porter stemming. On top of this, an additional ToughFilter (Figure 20) was implemented to eliminate all tokens not entirely composed of alphabetical letters. Altogether, the Gutenberg Analyzer reduced the number of tokens dramatically and allowed the algorithm to be run on the limited resources available to the cluster machines.

```
static class ToughFilter extends FilteringTokenFilter
      {
        private final CharTermAttribute termAtt = addAttribute(CharTermAttribute.class);

                public ToughFilter(Version version, TokenStream input)
                {
                        super(version, input);
                }

                @Override
                public boolean accept() throws IOException
                {
                        // Remove numbers and all tokens containing non-alpha characters.
                        return StringUtils.isAlpha(termAtt.toString());
                }
        }

        @Override
        protected TokenStreamComponents createComponents(String arg0, Reader reader)
        {
                Tokenizer tokenizer = new StandardTokenizer(Version.LUCENE_46,reader);
                TokenStream result = new StandardFilter(Version.LUCENE_46, tokenizer);
                 result = new ToughFilter(Version.LUCENE_46, result);
                 result = new LowerCaseFilter(Version.LUCENE_46, result);
                 result = new LengthFilter(Version.LUCENE_46, result, 3, 20);
                 result = new StopFilter(Version.LUCENE_46, result,
StandardAnalyzer.STOP_WORDS_SET);
                 result = new PorterStemFilter(result);
                return new TokenStreamComponents(tokenizer, result);
        }
```

***Figure 20**: ToughFilter and GutenbergAnalyzer*

With the resultant vectors small enough to be run, Table 8 displays the first 8 topics output by the algorithm.

	Topic #1	**Topic #2**	**Topic #3**	**Topic #4**	**Topic #5**	**Topic #6**	**Topic #7**	**Topic #8**
Word 1	cathol	tom	russia	vpon	israel	growth	unto	silver
Word 2	jesu	loui	princess	vol	jew	rate	thou	bronz
Word 3	sidenot	labor	parliament	haue	arab	expenditur	saith	medal
Word 4	shalt	irish	treati	le	territori	gdp	shalt	work
Word 5	pope	coloni	republ	caesar	egypt	econom	nephi	dutch

***Table 8**: LDA results*

Summary and Conclusions

The Hadoop framework was successfully utilized to perform text mining operations on two disparate textual datasets. The solutions engineered for the Leipzig Corpora dataset would translate well to a variety of small-format data stores such as Twitter analysis, etc. The analysis of the Project Gutenberg data harnessed cutting-edge algorithms to gain insight into a dataset composed largely of novels and other documents not traditionally used with clustering algorithms or topic modelling. While some of the CVB-generated topics were more coherent than others, the resource limitations of the cluster are perhaps more to blame than the algorithm. Though the machines were somewhat outdated hardware-wise, by combining them all together the Hadoop framework allowed analysis to be performed that would have been impossible otherwise.

Background and Related Work

This project utilizes a wide breadth of tools associated with the Hadoop framework that may not be familiar to Computer Scientists yet to encounter the platform. To better organize these brief introductions, they are henceforth categorized into two subsets: core Hadoop components and higher-level Hadoop "ecosystem" projects that make use of an underlying Hadoop architecture.

Hadoop Core Components: Hadoop

Hadoop is an open-source framework that aims to solve problems associated with large scale data storage and processing. The framework itself was based off of two papers released by Google: "The Google File System" in 2003 and "MapReduce: Simplified Data Processing on Large Clusters," released in 2004 [12, 13]. Google itself had been utilizing the described filesystem and MapReduce approach effectively to solve its own scalability problems and following the release of the papers, many other companies became interested in emulating Google's approach.

Apache Hadoop grew out of Apache Nutch, an open source web search engine. Following the publication of the Google papers on their file system and MapReduce algorithm, the creators of Nutch attempted to integrate the ideas put forth by Google into their own project. By early 2006 the Hadoop project was split off from Nutch to become an open source implementation of the Google file system and data processing methods without being tied to any specific project in particular [14].

Shortly after its release to the open source community, Yahoo! provided a dedicated team to work on the Hadoop project full time, attempting to turn the project into something that could truly run at "internet-scale." Many other companies quickly joined the fold as the project began to gain traction and perform well on many different benchmarks.

In the time since its creation, the Hadoop framework has been adopted by a who's who of large tech companies that deal with any sort of scalability issues: Facebook, Amazon, Rackspace, IBM, Oracle, Dell, Intel, Microsoft, Adobe, AOL, EBay, Hulu, Last.fm, LinkedIn, The New York Times, Spotify, Twitter, Yahoo!, and many, many more. It has become the go-to tool for handling big data issues and new applications built upon the framework are under constant development [15].

Hadoop Core Components: The Hadoop Distributed Filesystem

The Hadoop Distributed Filesystem (HDFS) is a central component of the Hadoop framework and forms the primary (and default) distributed storage mechanism used by Hadoop applications. HDFS is written in Java and designed to integrate well with the MapReduce paradigm. It is also designed to be fault tolerant, scalable, and easy to balance and expand.

Each HDFS cluster contains a namenode daemon process that is responsible for keeping track of where data blocks are located throughout the nodes of the cluster. Data itself is stored on datanode members, who coordinate with the central namenode to inform it of their file contents and replication responsibilities.

HDFS is fault tolerant because of its ability to mandate replication of data between datanodes such that if any one goes down, the data can always be retrieved from another node. In the interest of data locality, HDFS will attempt at first to co-locate data to a nearby datanode, so that the data can be transferred to it quickly and only a relatively small amount of cluster network bandwidth is consumed. Once a data block has been replicated to another rack-local datanode, HDFS can be configured to also backup the data block to another datanode not present on the same rack, to make sure that the data is protected even if the entire rack were to fail. This three-tiered replication is the default methodology employed by HDFS when operating in a distributed mode, though it can be configured higher (and to replicate even to other data centers, etc) or lower, in case resources are limited and the data being stored is not of any tremendous value [14].

Hadoop Core Components: MapReduce

MapReduce is another central component of the Hadoop framework. MapReduce itself functions as a programming model for processing vast amounts of data through a simple, parallel, distributed algorithm. The MapReduce implementation provided in the Hadoop framework can be utilized by a variety of different programming languages: Java by default, C++ with Hadoop Pipes, and Ruby, Python, BASH, etc. through Hadoop Streaming (communicates over STDOUT).

As the name implies, the MapReduce algorithm consists of two phases: a map phase and a reduce phase. Each of the phases take key-value pairs as input and also output key-value pairs. The specifics and types of these pairs are defined by the programmer.

Generally speaking, the Map() function performs filtering and sorting of the input data while the Reduce() function often performs some form of summary operation across the key-value pairs outputted by the Map() function.

Hadoop Core Components: YARN (Yet Another Resource Negotiator)

The Hadoop computation and execution framework underwent a considerable overhaul in the recently-released Hadoop 2 (MapReduce 2.x) releases. Prior to Hadoop 2 and the introduction of YARN, the MapReduce framework was the sole way provided by Hadoop to interact with cluster data. Though other kinds of solutions were often “hacked” into MapReduce format, there was often a considerable performance overhead.

In previous versions of MapReduce, a centralized JobTracker node kept track of which nodes (TaskTrackers) had empty Map or Reduce “slots” that were available. In addition to monitoring the available resources in the cluster, the JobTracker was responsible for scheduling and monitoring all jobs that were submitted to the cluster. On many large clusters, the dual roles played by the JobTracker began to test the scalability of the Hadoop architecture. In addition to the over-taxation of a singular JobTracker node, MapReduce's inherent speed limitations caused many engineers to seek a decoupling of Hadoop's processing capabilities from MapReduce as the sole means of computing over HDFS data.

More than simply separating the old JobTracker's responsibilities into two separate daemons (the new ResourceManager and ApplicationManager), YARN also abstracts the Hadoop architecture to allow many distributed applications other than just MapReduce to take advantage of cluster resources. Applications in YARN now can request cluster resources in the form of containers (a specified amount of RAM, CPU) from the ResourceManager, allowing YARN to act in many ways like a distributed operating system.

When a MapReduce application or any other YARN program is now launched, an ApplicationMaster is launched on the machine that submitted the job. The ApplicationMaster is responsible for coordinating with the ResourceManager concerning the application's execution on the cluster. If enough NodeManagers (running on all slave nodes) in the cluster indicate that they have the requisite resources to run the application, the ResourceManager will grant the ApplicationMaster container leases upon which it can execute tasks, optimizing whenever it can for locality to the data to be processed. Overall the addition of YARN to the Hadoop framework has made it much more flexible and scalable without sacrificing any performance on most benchmarks relative to Hadoop 1.

Hadoop Ecosystem Component: Hive

Hive is a data warehouse infrastructure built on top of Hadoop for providing easy data summarization, query, and analysis. It was initially developed at Facebook but now main contributors also include NetFlix and Amazon.

Hive works by letting the user define a schema with which to process a large data set. This schema was previously defined in HCatalog but now has been rolled into Hive itself. Once the schema is provided and the Hive system can find it in a specified traditional RDBMS, Hive provides an SQL-like query language called HiveQL that allows users inexperienced with the MapReduce/Hadoop framework to interact with the datasets through standard queries that are implicitly converted to MapReduce queries by the Hive libraries. Even for users fairly experienced with MapReduce, writing a custom MapReduce job often still requires a non-trivial amount of time and effort. Internally, Hive converts the submitted HiveQL queries into directed acyclic graphs of MapReduce jobs [4]. Unless the desired information is likely to be run extremely often or require some very eccentric features of the MapReduce architecture, Hive provides a relatively simple and intuitive way to learn from large aggregate datasets.

Hadoop Ecosystem Component: Pig

Pig was developed by Yahoo! Research in 2006 as a higher layer of abstraction for processing large datasets. Pig's main component is the Pig Latin language which lets users easily define a series of transformations applied to input data to produce output. It "feels" more like a traditional programming language with the notable exception of not providing any control flow operations; generally if such things are needed, a language such as Java can be utilized and then subsequently call the dependent Pig Latin statements therein when appropriate (all Hadoop components integrate fairly seamlessly with Java). Like HiveQL, Pig converts the Pig Latin statements into a series of MapReduce jobs.

What separates Pig from HiveQL is that it is designed to do a lot more than simple queries of the dataset. With few exceptions, if something can be done in MapReduce it can also be done via Pig Latin in far fewer lines of code. This includes things such as loading data, sorting, filtering, grouping, joining, moving, etc [5]. While Pig Latin's higher layer of abstraction does impose something of a performance penalty versus regular MapReduce, the gap has been narrowing considerably in recent years.

Hadoop Ecosystem Component: Mahout

Mahout is an extensive machine learning project from the Apache Software Foundation. Mahout began as a subproject of Apache's Lucene project, which is an open-source text search engine (Nutch grew out it also). The project focuses especially on distributed and scalable machine learning applications with its three primary functions currently being recommender engines (collaborative filtering), clustering, and classification.

Mahout is written entirely in Java and most algorithms (though not all) are built to utilize a Hadoop cluster [7]. The main project is under extremely active development and new algorithms and implementations are being added all the time. Though it does require data from Hadoop to be in very particular formats, Mahout provides a extremely useful and powerful tool for learning from the sort of large datasets commonly seen on Hadoop.

The Mahout project is still in the midst of being migrated to Hadoop 2 and no official releases currently support execution on YARN. The version utilized throughout the project is a pre-alpha development snapshot (12 Apr 2014) modified in several places by the author.

Hadoop Ecosystem Component: Lucene

Though not fundamentally tied to Hadoop, Apache Lucene is in some respects a maternal project, having spawned the Apache Nutch project that begot Hadoop. Lucene's primary purpose is full-text indexing and searching. At the core of Lucene's architecture is the concept of documents and the text fields contained therein. Following tokenization but prior to final indexing, a filter can be applied to the text data obtained in the document to weed out unwanted tokens matching a pattern. It continues to be designed with Hadoop applications in mind, ensuring that it's data can be easily integrated into the Hadoop framework [16].

Background: Hadoop benchmarks

A widely-publicized early example of Hadoop's power came in 2007, when the New York Times used Hadoop to sort through over four terabytes of archives and convert them to PDF files available over the web [17]. Not long after this example a Hadoop cluster broke the world record for sorting a terabyte of data. Running on 910 commodity nodes, Hadoop was able to sort a terabyte in under 3 and a half minutes [18]. In Novermer of 2008, Google shattered the terabyte sort record by using its proprietary MapReduce implementation to perform the sort in a mere 68 seconds. It also successfully sorted a petabyte of data in six hours and two minutes. The current record for terabyte sort on Hadoop appears to belong to MapR with a time of 54 seconds [19].

While the cluster utilized throughout this project would not be capable of tackling such massive sorts, the sorting benchmark remains an effective test of cluster power with smaller datasets (200GB used in this project) and is utilized extensively throughout the benchmarking section of the accompanying project user guide.

Related Work: Hadoop Tuning

While much prior work has been done in the field of high-performance tuning for Hadoop clusters [20, 21], very little has yet been released pertaining to the new YARN architecture. Hortonworks includes with their new distribution a python script intended to detect proper settings for some common properties and these were used as an initial baseline (improved settings run over 350% faster on sort benchmark) [22].

In addition to the absence of information regarding optimizations specific to the new YARN architecture there is also a dearth of information pertaining to running Hadoop clusters on somewhat “less than” commodity hardware. Currently in industry, a “commodity” Hadoop node in terms of hardware would be somewhere along the lines of 2+ quad-core 2.5+ GHz CPUs with 32-256 GB ECC RAM and 4+ 1TB+ SATA disks. Aspects of dealing with lower-grade hardware in an extremely hardware-heterogenous cluster have not been addressed.

Related Work: Using Mahout for clustering, classification of text data

Numerous published projects have used Mahout for textual analysis [23]. Many of the new Mahout features however have yet to be utilized in broadly-published examples and are in many cases on the cutting edge of NLP/statistical analysis.

The LDA algorithm was first presented as an algorithm for topic discovery in 2003 [10]. It has been utilized to categorize scientific journal articles by topic [24], but work has not been performed with less rigidly defined fictional works such as the Project Gutenberg text documents. Mahout added the Collapsed Variational Bayesian implementation of LDA in it's 0.8.0 release (2013-07-24) [25].

References

[1] SINTEF. (2013, May 22). Big Data, for better or worse: 90% of world's data generated over last two years. ScienceDaily. Retrieved January 27, 2014 from www.sciencedaily.com/releases/2013/05/130522085217.html

[2] Walter, Chip. "Kryder's law." *Scientific American* 293.2 (2005): 32-33.

[3] Biemann, Chris, et al. "The Leipzig Corpora Collection-monolingual corpora of standard size." *Proceedings of Corpus Linguistic 2007* (2007).

[4] White, Tom. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[5] C. Olston, B. Reed, U. Srivastava, R. Kumar and A. Tomkins, “Pig Latin: A Not-So-Foreign Language for Data Processing.”ACM SIGMOD 2008, June 2008.

[4] Ramos, Juan. "Using tf-idf to determine word relevance in document queries." In *Proceedings of the First Instructional Conference on Machine Learning*. 2003.

[5] Teh, Yee Whye, David Newman, and Max Welling. "A collapsed variational Bayesian inference algorithm for latent Dirichlet allocation." In *NIPS*, vol. 6, pp. 1378-1385. 2006.

[6] Thusoo, Ashish, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. "Hive: a warehousing solution over a map-reduce framework." *Proceedings of the VLDB Endowment* 2, no. 2 (2009): 1626-1629.

[7] Anil, Robin, Ted Dunning, and Ellen Friedman. *Mahout in action*. Manning, 2011.

[8] Strehl, Alexander, Joydeep Ghosh, and Raymond Mooney. "Impact of similarity measures on web-page clustering." In *Workshop on Artificial Intelligence for Web Search (AAAI 2000)*, pp. 58-64. 2000.

[9] Huang, Anna. "Similarity measures for text document clustering." In*Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008), Christchurch, New Zealand*, pp. 49-56. 2008.

[10] Blei, David M., Andrew Y. Ng, and Michael I. Jordan. "Latent dirichlet allocation." *the Journal of machine Learning research* 3 (2003): 993-1022.

[11] Foulds, James, Levi Boyles, Christopher DuBois, Padhraic Smyth, and Max Welling. "Stochastic collapsed variational Bayesian inference for latent Dirichlet allocation." In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 446-454. ACM, 2013.

[12] S. Ghemawat, H. Gobioff and S. Leung. “The Google file system,” In Proc. of ACM Symposium on Operating Systems Principles, Lake George, NY, (Oct 2003): pp 29–43.

[13] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters.” Communications of The ACM 51-1 (2008): 107–113.

[14] Borthakur, Dhruba. "The hadoop distributed file system: Architecture and design." *Hadoop Project Website* 11 (2007): 21.

[15] https://wiki.apache.org/hadoop/PoweredBy

[16] Lucene, Apache. "A high-performance, full-featured text search engine library."*URL: http://lucene. apache. org* (2005).

[17] Derek GottFrid, “Self-service, Prorated Super Computing Fun!” 1 November 2007, http://http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/

[18] “Sorting 1PB with Mapreduce,” 21 November 2008, http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html

[19] “MapR and Google Compute Engine Set New World Record for Hadoop TeraSort,” 24 October 2012, http://www.businesswire.com/news/home/20121024005285/en/MapR-Google-Compute-Engine-Set-World-Record.

[20] Joshi, Shrinivas B. "Apache hadoop performance-tuning methodologies and best practices." In *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering*, pp. 241-242. ACM, 2012.

[21] Herodotou, Herodotos. "Hadoop performance models." *arXiv preprint arXiv:1106.0940* (2011).

[22] Hortonworks.com “Determine YARN and MapReduce Configuration Settings.” URL: http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.0.6.0/bk_installing_manually_book/content/rpm-chap1-11.html (2014).

[23] Esteves, Rui Maximo, Rui Pais, and Chunming Rong. "K-means clustering in the cloud--a Mahout test." In *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*, pp. 514-519. IEEE, 2011.

[24] Griffiths, Thomas L., and Mark Steyvers. "Finding scientific topics."*Proceedings of the National academy of Sciences of the United States of America* 101, no. Suppl 1 (2004): 5228-5235.

[25] Mahout, Apache. "Scalable machine learning and data mining." (2012).

[26] https://github.com/szweibel/linked-gutenberg/blob/master/stripgutenberg.pl

Appendix A
Hadoop Installation Document

OS: CentOS 6.5-minimal with basic video driver.
Rationale: CentOS is a free, enterprise-class operating system with 100% binary compatibility with it's upstream source, Red Hat Enterprise Linux. RHEL is the OS used most in industry, so CentOS seemed like a great choice. With the exception of one node (so we can view Ganglia metrics, data visualizations, etc.), the minimal, non-GUI install was used in order to maximize the resources available to the Hadoop daemons.
Note: instructions cover setting up a GATEWAY IP on a dual-NIC machine so that nodes can communicate with themselves locally (192.168.1.#) as well as get out to the internet through one machine. If using a router instead of a switch, this won't be necessary.
Note #2: Hadoop clusters of even relatively moderate size can consume vast amounts of power when under full stress. Prior to setting up and benchmarking a cluster, make sure that there exists an adequate power supply to handle the cluster.

Install OS and setup 'hadoop' user:
1) Burn .iso image to USB (Win32 Disk Imager works well).
2) Escape to setup on BIOS load, specify to load/install from USB stick.
3) Follow CENTOS installation instructions.
4) Login as root and perform following:
- `useradd hadoop`
- `passwd hadoop`
- `visudo # give hadoop sudo privileges`

Post-Install:
1) Disable SELinux: set "`SELINUX=disabled`" in /etc/selinux/config
- `su`
- `echo 0 > /selinux/enforce`

2) To improve SSH speed, edit /etc/ssh/sshd_config and set all "GSSAPI" options to "no" and then restart the SSHD service:
- `sudo service sshd restart`

3) Disable IPv6 by editing /etc/sysctl.conf:
- `net.ipv6.conf.all.disable_ipv6 = 1`
- `net.ipv6.conf.default.disable_ipv6 = 1`

4) /proc/sys/net/core/somaxconn corresponds to the limit of socket listen() backlog. On CentOS, this value defaults to 128, which is way too low to handle the bursts of requests common in a Hadoop cluster. To raise it:
- `sudo/su vi /etc/sysctl.conf` -> add `net.core.somaxconn=1024`

5) By default, Linux keeps track of the last time any file was accessed (read, executed, etc). This means that each read access to a file will also feature a write access to update the last time it was accessed. For a Hadoop cluster, this is a performance killer. To turn it off:
- `sudo/su vi /etc/fstab` -> find the root filesystems ('/') you're using and in

the fourth column should be a “defaults” value. Change this “defaults” to “defaults, noatime” and exit the file.

- `mount -o remount / #` and any other partitions on multi-disk setups

6) To reload the sysctl interface: `sudo sysctl -p`

7) Nodes within a Hadoop cluster often deal with a substantial amount of open files and require a large amount of running processes. The CentOS/Linux default is 1024, something we should increase permanently. In /etc/security/limits.conf add these lines:

```
hadoop    hard nofile    65536
hadoop    soft nofile    65536
hadoop    hard nproc     65536
hadoop    soft nproc     65536
```

8) The edits in /etc/security/limits.conf are permanent but will not take effect in the current shell. To change the values in the running shell:

- `ulimit -n 4096`
- `ulimit -u 4096`
- `ulimit -a # to ensure settings “took”`

Set up networking on dual-NIC card machine:

- ensure “NETWORKING=yes” in /etc/sysconfig/network
- edit /etc/sysconfig/network-scripts/ifcfg-eth0 and set “ONBOOT=yes” and “BOOTPROTO=dhcp”
- edit /etc/sysconfig/network-scripts/ifcfg-eth1 and set “ONBOOT=yes”, “BOOTPROTO=static”, “IPADDR=192.168.1.1”, “NETMASK=255.255.255.0”
- `sudo service network restart`
- `sudo dhclient eth0`
- then shut down dhclient so it doesn’t later wipe the static IP.

Set up IP forwarding on dual-NIC card machine:

- execute “`iptables -A FORWARD -i eth1 -j ACCEPT`” to tell firewall to allow incoming FORWARD packets over the eth1 (internal) device interface.
- execute “`iptables -A FORWARD -o eth0 -j ACCEPT`” to tell firewall to allow outgoing FORWARD packets over the eth0 (external) device interface.
- edit /etc/sysctl.conf and set “`net.ipv4.ip_forward = 1`”
- enable the above change: “`sysctl -p /etc/sysctl.conf`”
- execute “`iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE`” to mask external requests from local LAN nodes with the IP address of our gateway. POSTROUTING just specifies that packets can be altered as they’re leaving the firewall’s networking device.
- add “`sudo iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE`” to ~/ipTemp.sh so that it doesn’t have to be run every time we reboot the computer.

Set up networking on single-NIC card machines:

- edit /etc/sysconfig/network-scripts/ifcfg-eth0 and set “ONBOOT=yes”, “BOOTPROTO=static”, “IPADDR=192.168.1.###” and “NETMASK=255.255.255.0” (can use another identifier instead of .1. if using multiple racks - reconfigure netmask).

- edit /etc/sysconfig/network and set “NETWORKING=yes”, “GATEWAY=<internal IP address of dual-NIC card machine>”
- add DNS information to /etc/resolv.conf (should match dual-NIC’s DHCP-generated resolv.conf):
    - `search appstate.edu`
    - `nameserver 152.10.2.222`
    - `nameserver 152.10.2.223`
- `sudo service network restart`
- ensure external IP matches dual-NIC machine by doing a ‘`curl http://myip.dnsomatic.com`’
- make sure dhclient is not running else it will periodically wipe the LAN ip: “`ps -eaf | grep dhclient`”
- `sudo yum remove NetworkManager`
- `sudo yum remove dhclient`

Set up passwordless SSH between all nodes:

- each node must generate an RSA key pair for authentication (as hadoop user): “`ssh-keygen -t rsa -f ~/.ssh/id_rsa`”
- enter passphrase that will be same for all nodes in the cluster.
- copy the generated public key back to a shared location: “`scp ~/.ssh/id_rsa.pub hadoop@192.168.1.1:~/.ssh/cluster/thisNode.pub`”
- disable StrictHostKeyChecking in /etc/ssh/ssh_config by setting “StrictHostKeyChecking no”
- “`sudo service sshd restart`”
- once all nodes have created RSA keys and copied them to a shared location, go to that location and append all public keys to the authorized_keys file: “`cat * >> ~/.ssh/authorized_keys`”
- edit the /etc/hosts file to include the IP addresses and hostnames of all machines in the cluster, as so:

```
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6
192.168.1.1 aho
192.168.1.2 tito
192.168.1.3 spino
192.168.1.4 nano
192.168.1.5 ammo
192.168.1.6 techno
192.168.1.7 dryo
192.168.1.8 grypo
192.168.1.9 anono
192.168.1.10 seismo
192.168.1.11 rhino
192.168.1.12 maino
192.168.1.13 newo
```

```
192.168.1.14 drapo
192.168.1.15 mino
192.168.1.16 mino
192.168.1.17 hippo
192.168.1.18 kepo
192.168.1.19 tonto
…
```

Figure A-1*: /etc/hosts*

- spread the authorized_keys file to all nodes in the cluster: "`scp ~/.ssh/authorized_keys 192.168.1.1:~/.ssh/`", etc. Once passwordless ssh is online we'll use rsync and other tools to make these common operations much less tedious.
- we need to set up ssh-agent to store/provide our passphrase upon SSH attempts. To do this, copy the clusterAdmin.sh script (documented further on in this document) into the home directory of the hadoop user and call it from the .bash_profile file (you can just copy over .bash_profile as well). It will handle setting up ssh-agent as well as provide an easier method to sync files across the cluster.
- SSH is extremely strict about permissions being set correctly. From hadoop user's home directory, do a "chmod -R 700 ~/", "`chmod -R 700 .ssh`", "`chmod 644 ~/.ssh/id_rsa.pub`" and "`chmod 600 ~/.ssh/id_rsa`"
- let's tell the OS to invoke clusterAdmin.sh automatically upon shell logins by adding the following to ~/.bash_profie:
  - `source clusterAdmin.sh` # should be in same dir as clusterAdmin.sh
  - make sure all nodes in the cluster get the modified .bash_profile; can SCP this from main node.

Hadoop Configuration:

Hadoop relies upon several configuration files for it's various daemons. These files can be found by default in /etc/hadoop/conf and consist of:

- core-default.xml -> system-wide settings (HDFS, YARN, etc).
- hdfs-site.xml -> settings pertaining only to the operation of HDFS.
- yarn-site.xml -> settings pertaining to the operation of YARN.
- mapred-site.xml -> settings pertaining to the operation of MapReduce on top of YARN.

The general format found within these files is:

```
<property>
    <name>#nameofproperty#</name>
    <value>#thevalue#</name>
</property>
```

With optional allowance for designating a property as `<final>true</final>` and including a `<description>` tag. Henceforth when setting these parameters, the notation:

- (core-site.xml) hadoop.tmp.dir -> /home/hdfs

Will refer to setting the hadoop.tmp.dir property found in /etc/hadoop/conf/core-site.xml in accordance with the above format to the value of "/home/hdfs"

Setting up Hadoop on dual-NIC node:

- install wget:
  - `sudo yum install wget`
- Get and install current Java JDK (replace specific .rpm with current JDK file):
  - `wget --no-cookies --no-check-certificate --header "Cookie: gpw_e24=http%3A%2F%24www.oracle.com" "http://download.oracle.com/otn-pub/java/jdk/7u51-b13/jdk-7u51-linux-x64.rpm"`
- Install the JDK:
  - `sudo rpm -ivh jdk-7u51-linux-x64.rpm`
- Set JAVA_HOME environment variable in .bash_profile:
  - `JAVA_HOME=/usr/java/jdk1.7.0_51/`
- Add the Apache Bigtop repo to the list of repos YUM manages:
  - `sudo "wget -O /etc/yum.repos.d/bigtop.repo http://www.apache.org/dist/bigtop/bigtop-0.7.0/repos/centos6/bigtop.repo"`
- Go get Hadoop/etc.:
  - `sudo yum install hadoop\* flume\* mahout\* oozie\* whirr\* hbase\* hive\* hue\* pig\* sqoop\*`
- Formant the HDFS namenode:
  - `sudo /etc/init.d/hadoop-hdfs-namenode init`
- Start the HDFS daemons:
  - `sudo service hadoop-hdfs-namenode start`
  - `sudo service hadoop-hdfs-datanode start`
- Initialize some HDFS idrectories for the daemon's we'll be running:
  - `sudo /usr/lib/hadoop/libexec/init-hdfs.sh`
- The defualt /tmp directory can be limitied in size by default by the OS (CentOS included). Let's change it:
  - (core-site.xml) hadoop.tmp.dir -> /home/hdfs
- Now make sure that the 'hdfs' user can write to the new temp directory:
  - `mkdir ~/hdfs`
  - `mkdir ~/hdfs/dfs`
  - `mkdir ~/hdfs/dfs/data`
  - `cd`
  - `sudo chown -R hdfs hdfs`
  - make sure 'hdfs' user has read/write/execute access in all directories on the way to the new data directory.
- We need to specify the directory we want YARN to write into on the HDFS filesystem when the ApplicationMaster is staging files:
  - (yarn-site.xml) yarn.app.mapreduce.am.staging-dir -> /user
- Specify addresses of Resourcemanager RPC ports:
  - (yarn-site.xml) yarn.resourcemanager.address -> $resourcemanager:8032
  - (yarn-site.xml) yarn.resourcemanager.resource-tracker.address -> $resourcemanager:8031

    - (yarn-site.xml) yarn.resourcemanager.admin.address -> $resourcemanager:8033
    - (yarn-site.xml) yarn.resourcemanager.scheduler.address -> $resourcemanager:8030
- CentOS automatically creates some disk partitions when it installs itself and by default limits the size of the '/' directory to 50GB, approximately. Just as we did with the /tmp directory, let's move the directory where YARN stores it's staging data and temporary map/reduce outputs:
    - (yarn-site.xml) yarn.nodemanager.local-dirs -> /home/hdfs/hadoop-yarn/cache/${user.name}/nm-local-dir
- Now we create the directory we indicated above:
    - `sudo mkdir -p /home/hadoop-yarn/cache/michael/nm-local-dir`
    - `sudo chown -R yarn /home/hadoop-yarn`
    - `sudo chgrp -R yarn /home/hadoop-yarn`
    - Periodically use df to monitor partition fullness during large M/R runs. Using something like "`sudo du /<dir> | sort -n -r | head -n 100`" can help find troublesome directories.
- When executing MapReduce jobs, we need to specify that they should run on our YARN cluster and not in pseduo-distributed or local mode:
    - (mapred-site.xml) mapreduce.framework.name -> yarn
- Start up the YARN daemons:
    - `sudo service hadoop-yarn-resourcemanager start`
    - `sudo service hadoop-yarn-nodemanager start`
- Ensure that the HDFS filesystem initialized correctly:
    - `hadoop fs -ls -R /`
- Set up HUE (edit /etc/hue/conf/hue.ini):
    - `secret_key=<30ish random characters>`
    - `in [[ mapred_clusters ]] [[[ default ]]], set "submit_to=False"`
    - `in [[ mapred_clusters ]] [[[ default ]]], set "thrift_port=9090"`
    - `in [[ mapred_clusters ]] [[[ default ]]], set "hadoop_mapred_home=/usr/lib/hadoop-mapreduce"`
    - `in [[ yarn_clusters ]] [[[ default ]]], set "hadoop_mapred_home=/usr/lib/hadoop-mapreduce"`
- to enable the hdfs web interface:
    - (hdfs-site.xml) dfs.webhdfs.enabled -> true
- in /etc/hadoop/conf/core-site.xml, add users 'hue', 'oozie', 'httpfs', 'yarn' and 'hadoop' to the list of proxyusers/proxygroups that can impersonate the proper HDFS permissions for arbitrary accesses/commands:
    - (core-site.xml) hadoop.proxyuser.hue.hosts -> *
    - (core-site.xml) hadoop.proxyuser.hue.groups -> *
    - (core-site.xml) hadoop.proxyuser.hadoop.hosts -> *
    - (core-site.xml) hadoop.proxyuser.hadoop.groups -> *

  - (core-site.xml) hadoop.proxyuser.oozie.hosts -> *
  - (core-site.xml) hadoop.proxyuser.oozie.groups -> *
  - (core-site.xml) hadoop.proxyuser.httpfs.hosts -> *
  - (core-site.xml) hadoop.proxyuser.httpfs.groups -> *
  - (core-site.xml) hadoop.proxyuser.yarn.hosts -> *
  - (core-site.xml) hadoop.proxyuser.yarn.groups -> *
- Add Linux supergroup ‘supergroup’ corresponding to default Hadoop supergroup:
  - `sudo groupadd supergroup`
  - `sudo usermod -a -G supergroup hadoop`
  - `hadoop fs -mkdir /user/hadoop`
- To deal with a conflict between the default HBASE REST API port and the Nodemanager, set the HBASE port to something else in /etc/hbase/conf:
  - (hbase-site.xml) -> hbase.rest.port -> 8070
- sudo service hbase-rest restart
  - Make sure it’s running on another port now:
  - `sudo netstat -tulpn | grep 8070`
    - look resulting pid up with “`ps -eaf | grep <pid>`”
- Add all Datanode/NodeManager slave machines to /etc/hadoop/conf/slaves on the master node so that the Hadoop daemons can remotely launch/kill the slave daemons on the remote machines.
- Now restart the nodemanager, which should no longer die because of the port conflict:
  - `sudo service hadoop-yarn-nodemanager start`
  - `sudo service hadoop-yarn-nodemanager status`
- Now let’s test that MapReduce can run on our new (single-node, currently) YARN setup:
  - `hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples*.jar pi 10 1000`

The clusterAdmin.sh Script:

The clusterAdmin.sh script contain a multitude of BASH functions that make operating the cluster significantly more convenient. First though it must be configured in accordance with the cluster it’s operating with.

- GATEWAY should be set to the dual-NIC GATEWAY node.
- NODES is an array containing all nodes present in the cluster.
- SLAVES is an array containing all nodes running just datanodes/nodemanagers.
- The various classes (CLASS_ACER, LOW_END, MID_END, MID_HIGH, APPO, NEWO, LARGE_HEAP) are node classes that will determine the Hadoop configuration files that are sync’ed to them. If you’re dealing with different hardware, it will be up to you to emulate the approach taken in this project with different hardware specifications.
- Also likely to need updating are the JDK version downloaded from Oracle and the Bigtop version taken from the bigtop repository.

Now that the script is configured, a brief description of it's functionality:

- install_node -> called on slave machines just finishing the initial set up procedures, the function automates that downloading and installation Java and Hadoop. Also sync's some configuration files from the master node (passed as parameter), makes necessary directories with proper permissions, installs some utility programs used during benchmarking, and starts the slave hadoop daemons.
- reformat_datanodes -> removes all VERSION information from datanodes prior to a namenode -format call.
- conf_sync -> for a certain specified node class, sync's a specified configuration setup to all members of the class. For example, conf_sync NODES:BASIC would distribute the configuration files found within the BASIC class to all nodes specified in the NODES array (which should be all of them).
- large_heap_sync -> after shutting down all cluster nodemanagers/datanodes, a call to this function will sync the large heap class of configuration files to the necessary nodes, reboot them and the resourcemanager, and allow for execution of tasks requiring a larger default Java heap space than some machines in the cluster could handle.
- conf_sync_all -> sync's normal configuration files to all the varying types of machines in the cluster.
- clear_logs -> clears the log directories of all nodes.
- execute_nodes -> will execute the string parameter command on all nodes in the cluster.
- admin_sync -> sync's the current node's admin files (/etc/host, .ssh/authorized_keys, .bash_profile, .bashrc, clusterAdmin.sh) to all nodes in the cluster and ensures proper permissions/owner are set.
- init_passphrases -> utilizes an accompanying expect script to set the ssh_agent passphrases for all members of the cluster. If the entire cluster goes down, the quickest way to properly get everything back to running as normal is execute this function (so nodes can communicate), followed by the reboot function.
- reboot -> reboots the Hadoop daemons for all machines in the cluster. Convenient to use following the cluster going down or Hadoop configuration files being updated.
- gateway_forward -> function called to properly initialize iptables for gateway node.
- solo_mode -> if tasks require a massive amount of heap space or can be run much faster uberized (within just one VM), it may be necessary/wise to only utilize the strongest node in the cluster. This function will stop all other nodemanagers and leave only the strongest node up, properly configured.
- undo_solo -> restores normal settings to the strongest node and reboots all nodes in the cluster.

Setting up Hadoop on other cluster nodes:

The clusterAdmin.sh script can be used to greatly simplify installation on slave nodes. Invoking the script with a (-i) flag and argument corresponding to the hostname of the master node will take care of the entire setup process.

- If TCP ack errors are encountered, iptables must be disabled or configured to accept the traffic between cluster nodes,

Set up Hive/HCatalog on dual-NIC node:
Hive/Beeswax must be configured to interact with a RDBMS that will store schema information for processing large jobs. MySQL provides a more powerful platform than the default derby setup and is necessary if Beeswax (the Hive WebUI) is to be used. To set up:
- sudo yum install mysql-server
- sudo service mysqld start
- sudo mysql_secure_installation

Once the root user has been registered, it is wise to add another hadoop-specific user:
- mysql -u root -p
- mysql> CREATE USER 'hadoop'@'localhost' IDENTIFIED BY 'appState';
- mysql> GRANT ALL PRIVILEGES ON *.* TO 'hadoop'@'localhost' WITH GRANT OPTION;
- mysql> CREATE USER 'hadoop'@'%' IDENTIFIED BY 'appState';
- mysql> GRANT ALL PRIVILEGES ON *.* TO 'hadoop'@'%' WITH GRANT OPTION;

Now edit /usr/lib/hive/conf/hive-site.xml:
- (hive-site.xml) javax.jdo.option.ConnectionURL -> jdbc:mysql://localhost/hiveMetastore?createDatabaseIfNotExist=true
- (hive-site.xml) javax.jdo.option.ConnectionDriverName -> com.mysql.jdbc.Driver
- (hive-site.xml) javax.jdo.option.ConnectionUserName -> hadoop
- (hive-site.xml) javax.jdo.option.ConnectionPassword -> <hadoopUserMySQLPassword>
- (hive-site.xml) hive.metastore.warehouse.dir -> /user/hive/warehouse

Hive/HCatalog needs to know how to interact with the RDBMS containing stored schem information. Since a MySQL database was used, a MySQL connector JAR is required.
- copy mysql-connector-java-<version>-bin.jar into /usr/lib/hive/lib/, /var/lib/sqoop/, and /usr/lib/hue/lib/ so that the various services installed on the dual-NIC master node can access MySQL.

Oozie will also need to know how to communicated with the MySQL database. The jar can be added to its classpath by placing it into HDFS at /user/oozie/share/hive/.
- hadoop fs -copyFromLocal /user/lib/hive/lib/mysql*.jar /user/oozie/share/hive/

Now that MySQL has been properly configured and the involved components have been configured to interact with it, restart the services:
- sudo service hive-server restart
- sudo service hive-metastore restart
- sudo service hive-webhcat-server
- sudo service sqoop-server restart
- sudo service hue restart

Extras:
View Cluster WebUI’s from outside firewall;
View Hadoop HTTP Daemons through external SSH tunnel:

- Connect to CS machine with (-Y) flag and X11 forwarding enabled locally.
- Launch FireFox and navigate to 'about:config'
- Change these settings:
  - `network.http.max-connections;42`
  - `network.http.max-connections-per-server;0`
  - `network.http.max-persistent-connections-per-proxy;0`
  - `network.http.max-persistent-connections-per-server;0`
  - `network.http.pipelining;true`
  - `network.http.pipelining.maxrequests;0`
  - `network.http.pipelining.ssl;true`
  - `network.http.proxy.pipelining;true`

Maven:
At least one member of the cluster will need maven installed in order to (re)-build the project:

- `wget http://repos.fedorapeople.org/repos/dchen/apache-maven/epel-apache-maven.repo -O /etc/yum.repos.d/epel-apache-maven.repo`
- `yum install apache-maven`
- `sudo ln -s /usr/share/apache-maven/bin/mvn /usr/bin/mvn`

Direct SCP from computer outside firewall into cluster:

- `scp -r -o 'Host vizio.cs.appstate.edu' -o 'ProxyCommand ssh username@cs.appstate.edu nc %h %p' <file/directory> hadoop@vizio.cs.appstate.edu`

Appendix B
Data Preparation Document

Prepare Leipzig Corpora data:
The Leipzig Corpora databases can be downloaded from http://corpora.uni-leipzig.de/download.html. They should be stored together in a directory. The MySQL default data directory can then be changed to this directory by editing /etc/my.cnf (make sure mysql user has all privileges on new directory, or change mysql user to the hadoop user).

Once a directory is chosen, the data must be then imported into HDFS. Depending on the specific format desired, there are several methods to do this. To import the data in large, plain-text format under HDFS directories corresponding to the database of origin, the AutoSqoop component project can be used. The interface to this component is as follows:

```
Usage:
 [--mySqlName <<name>> --sqlTable <<table>> --tableKey <<key>> --input
<<dbDir>> --output <<hdfsDir>> --columns <<col(s)>> --seqFileFormat
--sqoopServer <<server>>]
Options
  --mySqlName (-n) <name>        MySQL username to use
  --sqlTable (-t) <table>        SQL Table to import
  --tableKey (-k) <key>          Primary key in table
  --input (-i) <dbDir>           Input directory (defaults to /var/lib/mysql)
  --output (-o) <hdfsDir>        Output directory (defaults to
                                 /user/$USERNAME/sqoopOut/)
  --columns (-c) <col(s)>        Comma-separated list of columns to import
  --seqFileFormat (-s)           Store on HDFS in SequenceFile format
  --sqoopServer (-a) <server>    Address of Sqoop RPC server
```
***Figure B-1**: AutoSqoop interface*

So once the desired Leipzig databases can be found in the MySQL directory (and all dbs belong to the mysql user and group), the AutoSqoop importation tool can be invoked lin a manner such as:

```
sudo hadoop jar TextProject-0.0.1-jar-with-dependencies.jar autosqoop -n hadoop -t
sentences -k s_id -i /var/lib/mysql -o /user/hadoop/sqoopOut/ -c sentence
```

Where the sentence field from all sentences tables found within databases stored in the /var/lib/mysql folder are imported to the /user/hadoop/sqoopOut/ folder in HDFS.

To import as SequenceFiles:

Mahout does not work well with plain-text data. While Sqoop2 is convenient and offers a rich amount of options for data importation and formatting, it lacks the ability to import text data in an appropriate Key,Value SequenceFile format. While it is capable of importing data as SequenceFiles, all contained data is stored in the Key with a NullWritable value type.

To get around this limitation, the SQL2Seq project was created. Given data stored in SQL databases similar to the Leipzig Corpora, the project will output to a specified HDFS directory Text <key,value> SequenceFiles of the format Key:<db_name><index>,Value:<sentence>.

The interface to the project is as follows:

```
Usage:
 [--mySqlName <mysqlName> --sqlTable <sqlTable> --input <input> --output
<output> --columns <columns>]
Options
  --mySqlName (-n) mysqlName    MySQL username to use
  --sqlTable (-t) sqlTable      SQL Table to import
  --input (-i) input            Input directory (defaults to /var/lib/mysql)
  --output (-o) output          Output directory (defaults to
                                /user/$USERNAME/sqoopOut/)
  --columns (-c) columns        Comma-separated list of columns to import
```

Figure B-2*: SQL2Seq interface*

So to import the desired Leipzig databases this time in a SequenceFile format more appropriate for Mahout (and still perfectly useable for Hive/Pig/MapReduce -- see SeqTableLoad.hql vs. TableLoad.hql), the following command can be used:

```
sudo hadoop jar TextProject-0.0.1-jar-with-dependencies.jar sql2seq -n root -t sentences
-i /var/lib/mysql -o /user/michael/SeqOut/ -c sentence
```

Where the sentence column from the sentences table is again imported from all matching databases found in input directory /var/lib/mysql and output to HDFS directory /user/hadoop/SeqOut/. The name of the output files within the specified directory will correspond to the current sentence index for the job at large.

Installing the Leipzig Corpora CorpusBrowser:

The Leipzig Corpora CorpusBrowser is an excellent data visualization and analysis tool for analyzing the individual Corpora databases one-by-one. It is useful not only for the ease of various small-level data explorations but also as an initial sanity check for results acquired through Hadoop. It is recommended to first perform new Hadoop-based queries on one component database which can then be verified against the CorpusBrowser results before performing the operation on the entire dataset.

The Browser can be downloaded in .zip form from http://corpora.uni-leipzig.de/download.html. Once downloaded and extracted, the CorpusBrowser.jar file can be executed immediately. A valid MySQL username and password is required to connect to the databases, and once provided the CorpusBrowser must be configured as to which databases to maintain connections to. Clicking “Settings” in the upper-right corner of the GUI will bring up an admin interface where all the databases should be entered. To greatly simplify this task, the CSV output of the ‘ls -m’ command (performed inside the database directory) can be copied into this field.

Once the CorpusBrowser has established connections to the desired databases, the databases can be queried for certain terms through the “Search” box. When enough data can be found for the term, a frequency count, list of commonly co-occurring words, and common left and right neighbors is displayed along with a graphical representation of other strongly co-occurring terms.

Prepare Project Gutenberg data:
The second dataset employed in this project is the Project Gutenberg collection found at http://www.gutenberg.org/. To sync the entire repository, rsync can be utilized:

rsync -av --del ftp@ftp.ibiblio.org::gutenberg /var/www/gutenberg

To download just archives containing English language text-formatted documents, extract them, and remove the Project Gutenberg headers, the script (getGutenberg.sh) can be used:

```
#!/bin/bash
wget -w 2 -m -H http://www.gutenberg.org/robot/harvest?filetypes[]=txt&langs[]=en
wait
find . -name "*.zip" -exec unzip -d EngText {} \;
for book in EngText/*.txt
do
    cat $book | stripgutenberg.pl > $book
done
mkdir FinalText
find ./EngText/ -name *.txt -size +50k -exec cp {} ./FinalText/ \;
rm -rf EngText
```

***Figure B-3**: getGutenberg.sh*

Where stripgutenberg.pl is Xavier Antoviaque’s perl script to scrub (many) of the inconsistently-formatted Project Gutenberg headers and footers. The final part of the above script checks for files below a certain size (likely those whose headers/footers were not recognized and stripped) and removes them from the final dataset. The header/footer removal script can found at https://github.com/antoviaque/language/blob/master/bin/stripgutenberg.pl.

To import large data files into HDFS without being copied to the local filesystem, the following process can be used:

```
wget http://student.cs.appstate.edu/kepplemr/66.tar.gz -O - | hadoop fs -put -
/user/michael/$(uuidgen).txt
```

Once the Gutenberg data has been downloaded, extracted and stripped, it can be copied directly into HDFS:

```
hadoop fs -put EngText
```

This plain-text format can be tied directly into the KMeans and LDA project implementations, which will be default (override with -s1 flag) generate properly-formatted SequenceFiles as the first step of their analysis. For example,

```
hadoop jar TextProject-0.0.1-jar-with-dependencies.jar lda -i EngText/ -o Eng_Out/ -maxDf
```

```
40 -minDf 1 -k 20 -size 15982369
```

Or if the directory is already in proper SequenceFile format:

```
hadoop jar TextProject-0.0.1-jar-with-dependencies.jar lda -i EngText/ -o Eng_Out/ -maxDf
40 -minDf 1 -s1 -k 20 -size 15982369
```

Appendix C
Cluster Benchmark Document

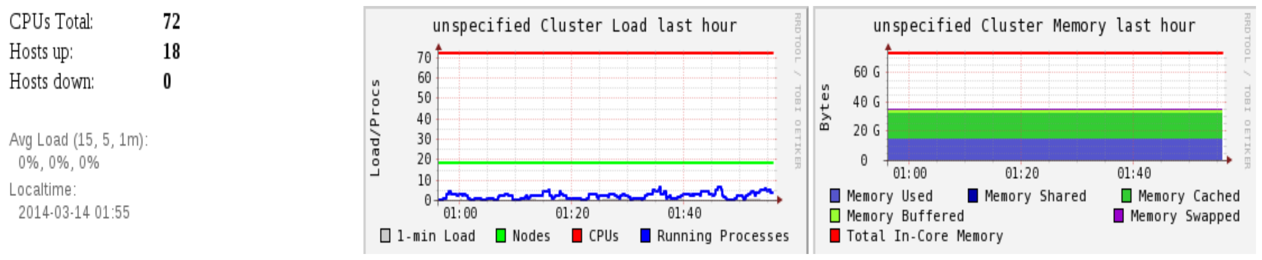
As a Hadoop cluster is brought online and optimized for peak performance, it is important to periodically benchmark the performance of the cluster to ensure any changes made were in fact beneficial to the performance of the cluster.

Install distributed monitoring software:
To assist in this task, the Ganglia distributed system monitoring software can be utilized. The one cluster node installed with a GUI can act as the main Ganglia monitoring node. To easily install the software needed, it is necessary to add the RPMForge repository:

- `wget . http://apt.sw.be/redhat/el6/en/x86_64/rpmforge/RPMS/rpmforge-release-0.5.3-1.el6.rf.x86_64.rpm`
- `rpm -ivh rpmforge-release-0.5.3-1.el6.rf.x86_64.rpm`
- Once the RPMForge repository is accessible, the main Ganglia packages can be installed:
    - `sudo yum install rrdtool ganglia ganglia-metad ganglia-gmond ganglia-web httpd`
    - `sudo service gmond start`
    - `sudo service gmetad start`
    - `sudo service httpd start`
- If everything went successfully, a Ganglia monitor should be visible by going to http://localhost/ganglia.
- To get Ganglia up and running on the other nodes, SCP the RPMForge rpm to those machines and install it as above. Then:
    - `sudo yum install ganglia ganglia-gmond`
    - `sudo service gmond start`

By default, Ganglia will monitor processor load, memory, network bandwidth and hard disk space, though it can be configured to monitor much more (lm-sensor CPU temperature information, etc).

Initial metrics at rest:
After installing the Ganglia monitoring software, the cluster monitor UI shows these basic metrics while at rest:

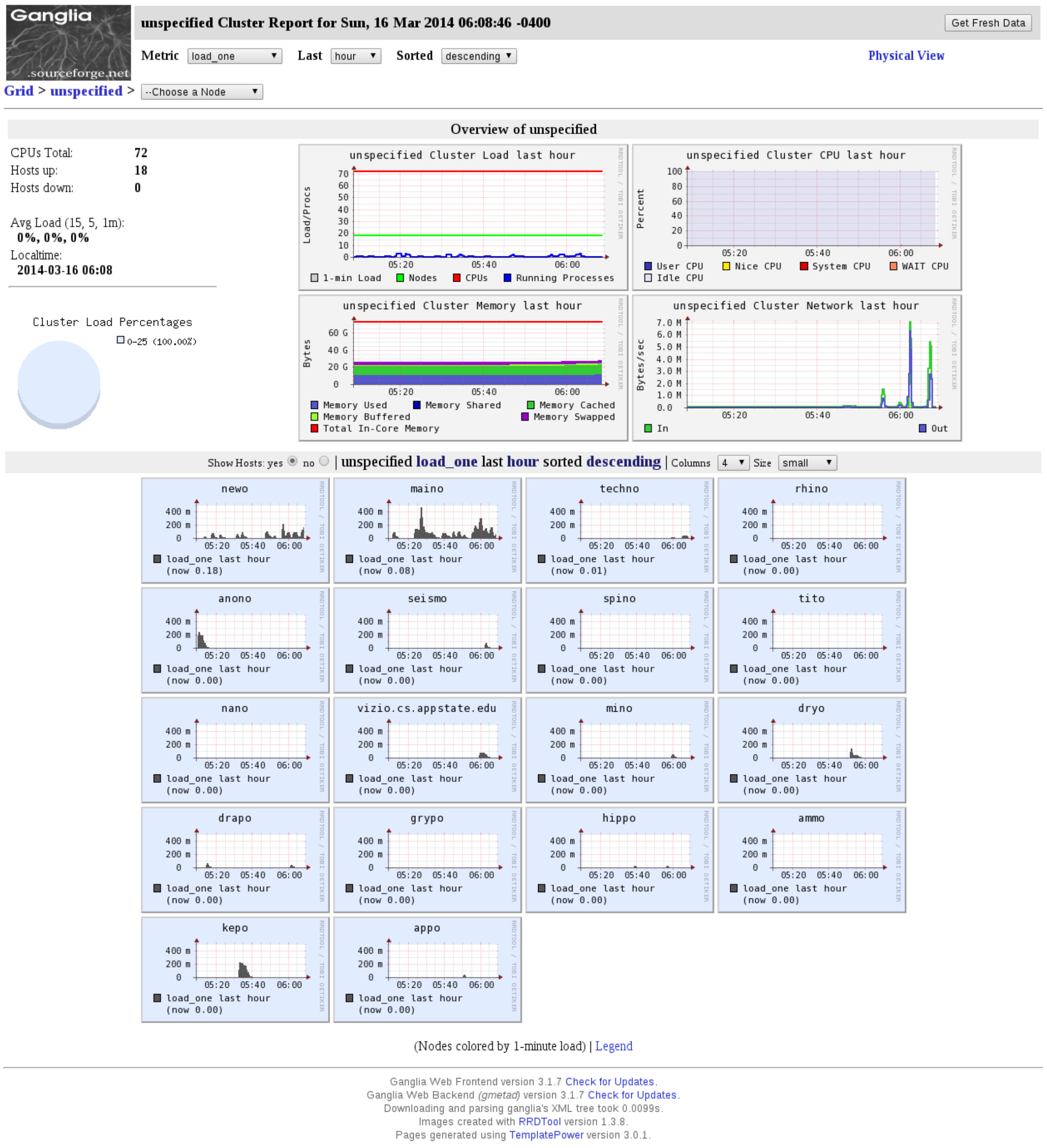


CPUs Total:	72
Hosts up:	18
Hosts down:	0

Avg Load (15, 5, 1m):
0%, 0%, 0%
Localtime:
2014-03-14 01:55

Figure C-1: *Initial Ganglia screen*

Clicking on the graphs will allow for expanded visualization and customization options, appearing initially like this:



unspecified Cluster Report for Sun, 16 Mar 2014 06:08:46 -0400

Metric load_one Last hour Sorted descending

Grid > unspecified > --Choose a Node

Overview of unspecified

CPUs Total: 72
Hosts up: 18
Hosts down: 0

Avg Load (15, 5, 1m):
0%, 0%, 0%
Localtime:
2014-03-16 06:08

Show Hosts: yes no | unspecified load_one last hour sorted descending | Columns 4 Size small

(Nodes colored by 1-minute load) | Legend

Ganglia Web Frontend version 3.1.7 Check for Updates.
Ganglia Web Backend (gmetad) version 3.1.7 Check for Updates.
Downloading and parsing ganglia's XML tree took 0.0099s.
Images created with RRDTool version 1.3.8.
Pages generated using TemplatePower version 3.0.1.

Figure C-2: Expanded Ganglia screen

Machine 'classes':

Proper configuration of a Hadoop cluster is essential to it's achieving optimal performance. In many/most Hadoop clusters, relatively homogenous hardware is used, making it relatively easy to assign configuration specifications across the cluster. In this case, the cluster is made up of very different hardware specs. To gain these specs, the following Linux commands can be used (dmidecode and hdparm may have to be installed: `sudo yum install hdparm`, `sudo yum install dmidecode`):

- free (see total and unused memory) or ‘vmstat -s -S -M | grep mem’ for more detailed breakdown.
- dmidecode --type 17 (determine memory speed)
- df (see size of disk/partitions)
- sudo hdparm -tT <mounted hard disk> (test speed of cached reads/buffered disk reads).
- nproc (see number of processors - includes SMT/Hyperthreaded cores).
- cat /proc/cpuinfo (see speed of processors)
- fdisk -l (check number of physical hard disks)

Hardware specifications for cluster nodes used in project:
1) Aho (Dual-NIC, Namenode)
- Memory: 4.1 GB @ 667 MHz (622 MB free)
- Disk: 520GB @ 3192.25 / 83.32
- Processors: 8 Xeon @ 1.86 GHz

2) Tito
- Memory: 4.0 GB @ 1066 MHz (3.1 GB free)
- Disk: 184GB @ 7557.01 / 88.88
- Processors: 2 Core 2 Duo @ 3 GHz

3) Spino
- Memory: 2.6 GB @ 800 MHz (2.15 GB free)
- Disk: 98GB @ 2597.25 / 70.52
- Processors: 4 AMD Phenom Quad-Core 9350e @ 2 GHz

4) Nano
- Memory: 2.6 GB @ 800 MHz (2.12 GB free)
- Disk: 98GB @ 2282.11 / 71.18
- Processors: 4 AMD Phenom Quad-Core 9350e @ 2 GHz

5) Ammo
- Memory: 2.6 GB @ 800 MHz (2.14 GB free)
- Disk: 98GB @ 2432.68 / 72.19
- Processors: 4 AMD Phenom Quad-Core 9350e @ 2 GHz

6) Techno
- Memory: 2.6 GB @ 800 MHz (2.13 GB free)
- Disk: 98GB @ 2418.30 / 71.93
- Processors: 4 AMD Phenom Quad-Core 9350e @ 2 GHz

7) Dryo
- Memory: 2.6 GB @ 800 MHz (2.12 GB free)
- Disk: 98GB @ 2679.21 / 71.88
- Processors: 4 AMD Phenom Quad-Core 9350e @ 2 GHz

8) Grypo
- Memory: 2.6 GB @ 800 MHz (2.08 GB free)
- Disk: 98GB @ 2437.58 / 72.23
- Processors: 4 AMD Phenom Quad-Core 9350e @ 2 GHz

9) Anono
- Memory: 2.6 GB @ 800 MHz (2.12 GB free)

- Disk: 98GB @ 2487.57 / 71.31
- Processors: 4 AMD Phenom Quad-Core 9350e @ 2 GHz

10) Seismo

- Memory: 2.6 GB @ 800 MHz (2.10 GB free)
- Disk: 98GB @ 2423.35 / 75.01
- Processors: 4 AMD Phenom Quad-Core 9350e @ 2 GHz

11) Rhino

- Memory: 1.9 GB @ 667 MHz (1.17 GB free)
- Disk: 97 GB @ 3292.20 / 42.23
- Processors: 8 Intel Xeon @ 2 GHz

12) Maino

- Memory: 3.84 GB @ 1333 MHz (3.05 GB free)
- Disk: 184 GB @ 6155.0 / 95.95
- Processors: 4 Intel Core i5 @ 3.2 GHz

13) Newo

- Memory: 16.27 GB @ 1600 MHz (14.25 GB free)
- Disk: 901 GB @ 16258.23 / 182.52
- Processors: 8 Intel Core i7 @ 3.4 GHz

14) Appo

- Memory: 7.92 GB @ 1066 MHz (7.17 GB free)
- Disk: 93 GB @ 2201.58 / 70.69
- Processors: 2 Intel Core 2 Duo @ 2.9 GHz

15) Drapo

- Memory: 3.84 GB @ 667 MHz (3.1 GB free)
- Disk: 97 GB @ 1283.59 / 70.08
- Processors: 2 Intel Core 2 @ 2.13 GHz

16) Mino

- Memory: 5.85 GB @ 667 MHz (5.12 GB free)
- Disk: 200 GB @ 4181.61 / 74.89
- Processors: 2 Intel Core 2 Duo @ 2.33 GHz

17) Hippo

- Memory: 3.79 GB @ 667 MHz (3.01 GB free)
- Disk: 338 GB @ 3974.09 / 75.88
- Processors: 2 Intel Core 2 Duo @ 2.33 GHz

18) Kepo

- Memory: 3.79 GB @ 667 MHz (3.02 GB free)
- Disk: 97 GB @ 3988.78 / 74.36
- Processors: 2 Intel Core 2 Duo @ 2.33 GHz

19) Tonto

- Memory: 5.85 GB @ 667 MHz (4.91 GB free)
- Disk: 200 GB @ 4181.61 / 74.89
- Processors: 2 Intel Core 2 Duo @ 2.33 GHz

Based upon these hardware specifications, the machines in the cluster could generally be divided up into six classes in order to make configuration simpler:

ACER_CLASS: Spino, Nano, Ammo, Techno, Dryo, Grypo, Anono, Seismo, Maino
LOW_END: Aho, Rhino
MID_RANGE: Tito, Drapo, Hippo, Kepo
NEWO: Newo
APPO: Appo
MID_HIGH: Mino, Tonto

The clusterAdmin.sh script will make it easy to update and push out configurations to the various classes of machines, but first the cluster will should be tested without any optimizations (other than those necessary to avoid errors) to get a baseline indication of performace:

HDFS Testing:
The most straightforward initial test for a cluster is to measure the performance of the HDFS filesystem on reads and writes. To accomplish this, the TestDFSIO program that comes bundled with Hadoop can be utilized (/usr/lib/hadoop-mapreduce/hadoop-mapreduce-*test*.jar).

- For these initial read/write tests, all meaningful work is done during the map stage and thus this stage is the only important one. A small subsequent reduce step merely aggregates the results from the various map tasks.
- The number of maps that will launch corresponds to the number of files to write/read (indicated by the '-nrFiles' flag).
- `time hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-*test*.jar TestDFSIO -write -nrFiles 10 -fileSize 1000`
- `time hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-*test*.jar TestDFSIO -read -nrFiles 10 -fileSize 1000`
- `hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-*test*.jar TestDFSIO -clean`

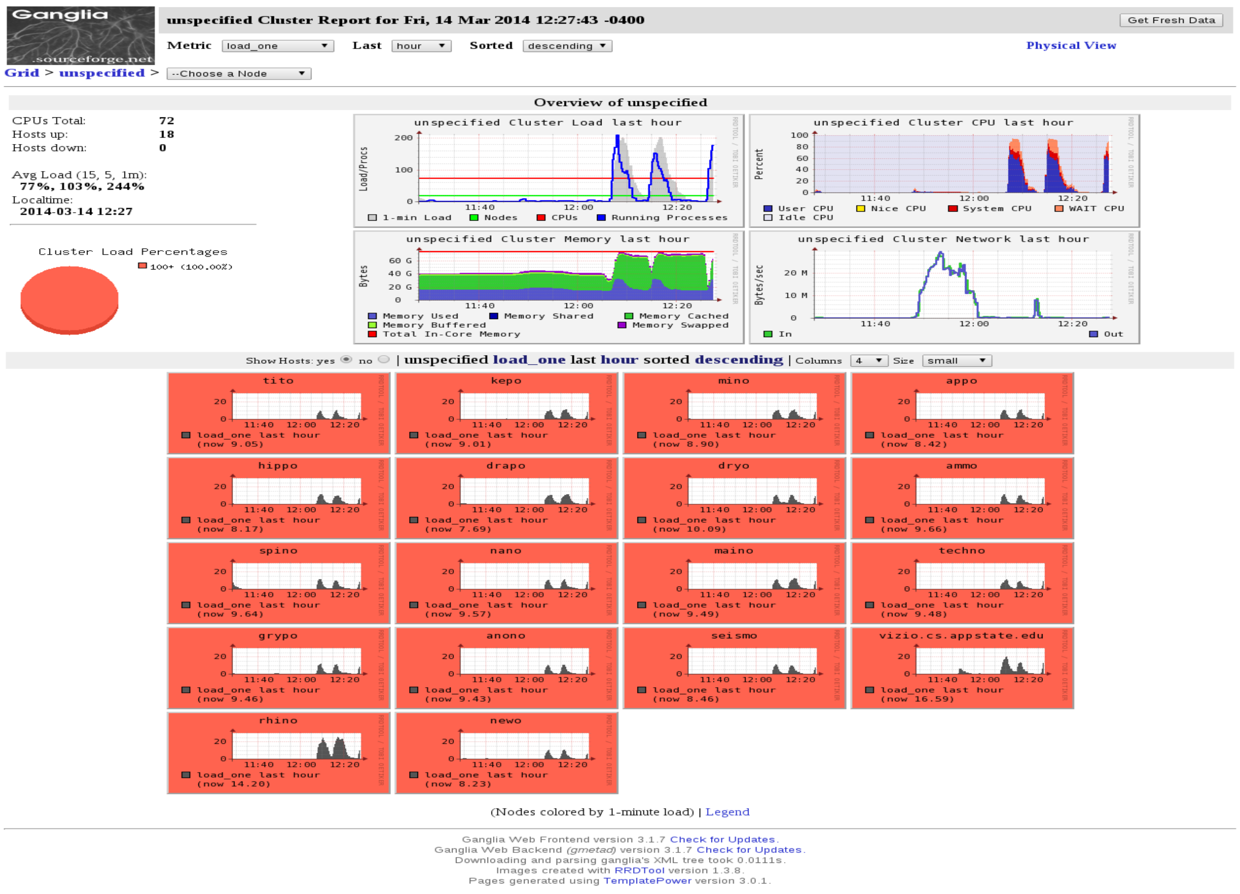
HDFS Initial Results:

- Test Write: `2:53.27`
- Test Read: `12:35.38`

The next logical step is testing the performance of MapReduce on YARN. To accomplish this on the cluster used throughout the project, 190GB+ data was generated and sorted. To easily generate such vast data, the randomwriter utility included in hadoop-examples can be used:

- To set the number of maps to be used, pass argument for test.randomwriter.maps_per_host:
    - `hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples-2.0.6-alpha.jar randomwriter -Dtest.randomwriter.maps_per_host=10 randomdata`

- To see how large the generated randomdata directory is:
  - `hadoop fs -du -s /user/hadoop/randomdata`
- When the MapReduce application deploys, it will indicate how many maps it will attempt to launch, “Running 190 maps,” (10 per node), which should reflect the argument set above. You can also use the -conf flag to specify an xml configuration file to pass to randomwriter.
- This test presents the first opportunity thus far to see the cluster under considerable stress, as indicated by the Ganglia monitoring:



unspecified Cluster Report for Fri, 14 Mar 2014 12:27:43 -0400

Metric load_one | Last hour | Sorted descending | --Choose a Node

Overview of unspecified

CPUs Total: 72
Hosts up: 18
Hosts down: 0

Avg Load (15, 5, 1m):
77%, 103%, 244%
Localtime:
2014-03-14 12:27

Show Hosts: yes no | unspecified load_one last hour sorted descending | Columns 4 | Size small

tito: load_one last hour (now 9.05)	kepo: load_one last hour (now 9.01)	mino: load_one last hour (now 8.90)	appo: load_one last hour (now 8.42)
hippo: load_one last hour (now 8.17)	drapo: load_one last hour (now 7.69)	dryo: load_one last hour (now 10.09)	ammo: load_one last hour (now 9.66)
spino: load_one last hour (now 9.64)	nano: load_one last hour (now 9.57)	maino: load_one last hour (now 9.49)	techno: load_one last hour (now 9.48)
grypo: load_one last hour (now 9.46)	anono: load_one last hour (now 9.43)	seismo: load_one last hour (now 8.46)	vizio.cs.appstate.edu: load_one last hour (now 16.59)
rhino: load_one last hour (now 14.20)	newo: load_one last hour (now 8.23)		

(Nodes colored by 1-minute load) | Legend

Ganglia Web Frontend version 3.1.7 Check for Updates.
Ganglia Web Backend (gmetad) version 3.1.7 Check for Updates.
Downloading and parsing ganglia's XML tree took 0.0111s.
Images created with RRDTool version 1.3.8.
Pages generated using TemplatePower version 3.0.1.

Figure C-3: Ganglia metrics when under heavy load

Now to sort the generated data, a sort program found can be used that is also located in the same examples JAR as used above. The number of reducers by default is 1.0 x configured capacity and may run faster with more.

- `time hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples-2.0.6-alpha.jar sort randomdata sorteddata`
- Since this sorting is by lengths the longest test we’ve performed so far, it is usually easier to deploy the three instances through a BASH script.
  - If benchmark parameters being tested are job-specific (mapred-site.xml, some YARN parameters), the TextProject project benchmark utility class can be used to configure such settings, execute sort tests, and aggregate results.
    - nohup hadoop jar TextProject-0.0.1-jar-with-dependencies.jar lda -i

Eng1000/ -o Eng1000_Out/ -maxDf 45 -minDf 1 -s1 -s2 -s3 -k 25 -size 5995645

```
#!/bin/bash
hadoop fs -rm -r /user/hadoop/sorteddata > sortOutput
time hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples-2.0.6-alpha.jar sort
randomdata sorteddata >> sortOutput
hadoop fs -rm -r /user/hadoop/sorteddata >> sortOutput
time hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples-2.0.6-alpha.jar sort
randomdata sorteddata >> sortOutput
hadoop fs -rm -r /user/hadoop/sorteddata >> sortOutput
time hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples-2.0.6-alpha.jar sort
randomdata sorteddata >> sortOutput
hadoop fs -rm -r /user/hadoop/sorteddata >> sortOutput
```

***Figure C-4**: Sort test script*

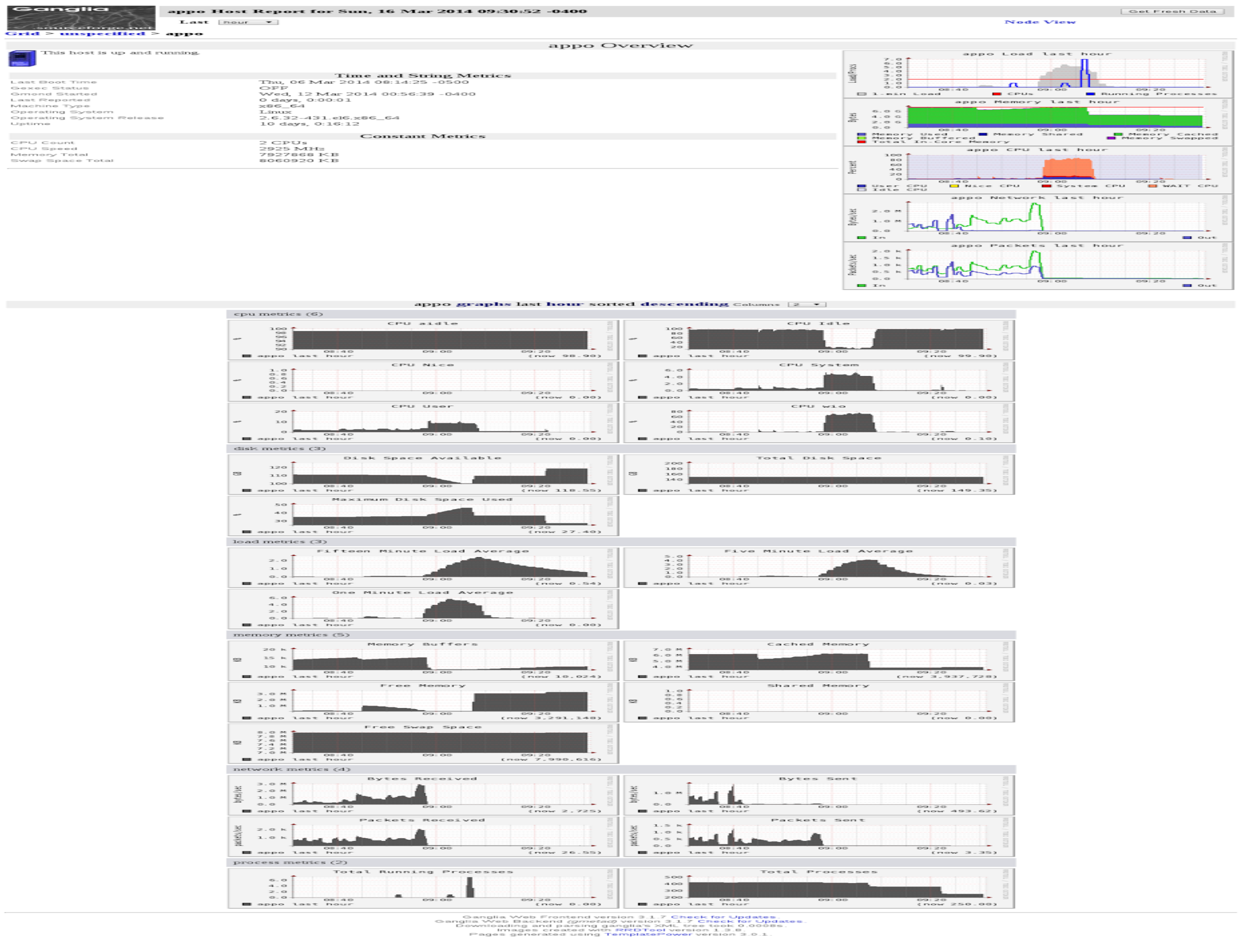
- Log4j sends most logging info to the console by default, leaving STDOUT with just the valuable run/time data:

```
Running on 36 nodes to sort from
hdfs://aho:8020/user/hadoop/randomdata into
hdfs://aho:8020/user/hadoop/sorteddata with 64 reduces.
Job started: Tue Mar 25 00:27:18 EDT 2014
Job ended: Tue Mar 25 02:16:18 EDT 2014
The job took 6539 seconds.
```

At this point, sorts are very unlikely to run without error. Many of the default configuration values do not suit the cluster well at all and others may prevent some nodes from running certain tasks entirely. To see what went wrong with running/failed containers, it is helpful to enable `yarn.log-aggregation-enable` and look in the directory specified `by yarn.nodemanager.remote-app-log-dir.` If remote logging is not enabled or HDFS is not functioning properly, check `yarn.nodemanager.log-dirs` (default to /var/lib/hadoop-yarn/containers on the local filesystem).

- copying the resulting log files to the local machine can make life easier, and make it simple to use Linux utils such as ‘grep’ to help identify the errors.
    - `hadoop fs -copyToLocal /var/log/hadoop-yarn/apps/hadoop/logs/application_1394842973416_0032 .`
    - `grep -R ERROR *`
- alternatively, Hadoop provides a cat implementation for HDFS:
    - hadoop fs -cat /var/log/hadoop-yarn/apps/hadoop/logs/application_`1394842973416_0032/* | grep “60000” | grep -v “600000”`
- The first error encountered during project testing was the ApplicationManager killing a task that failed to complete in 60 seconds, which is too little time for some of the weaker nodes. To fix this, alter mapred-site.xml:
    - (mapred-site.xml) mapreduce.task.timeout -> 1800000
- Also encountered are errors related to cleanup when a task is killed on a node (likelly due to speculative execution on another faster node concluding before the

node in question could finish). Setting this property will fix these:
  - (yarn-site.xml) yarn.app.mapreduce.am.job.committer.cancel -> 1800000
- Finally, some disk merge errors appeared following resolution of the above issues. This is often due to running out of disk space on a datanode slave. By examining the logs as above, it was determined that the error occurred on the node drapo, which can then be analyzed individually through the Ganglia monitor:



appo Host Report for Sun, 16 Mar 2014 09:30:52 -0400

Last hour

Grid > unspecified > appo

appo Overview

This host is up and running.

Time and String Metrics	
Last Boot Time	Thu, 06 Mar 2014 08:14:25 -0500
Gexec Status	OFF
Gmond Started	Wed, 12 Mar 2014 00:56:39 -0400
Last Reported	0 days, 0:00:01
Machine Type	x86_64
Operating System	Linux
Operating System Release	2.6.32-431.el6.x86_64
Uptime	10 days, 0:16:12

Constant Metrics	
CPU Count	2 CPUs
CPU Speed	2925 MHz
Memory Total	7927868 KB
Swap Space Total	8060920 KB

Ganglia Web Frontend version 3.1.7 Check for Updates.
Ganglia Web Backend (gmetad) version 3.1.7 Check for Updates.
Downloading and parsing ganglia's XML tree took 0.0008s.
Images created with RRDTool version 1.3.8.
Pages generated using TemplatePower version 3.0.1.

Figure C-5: Node diagnostics

- The Ganglia data indicates that the node ran out of disk space while under heavy load:

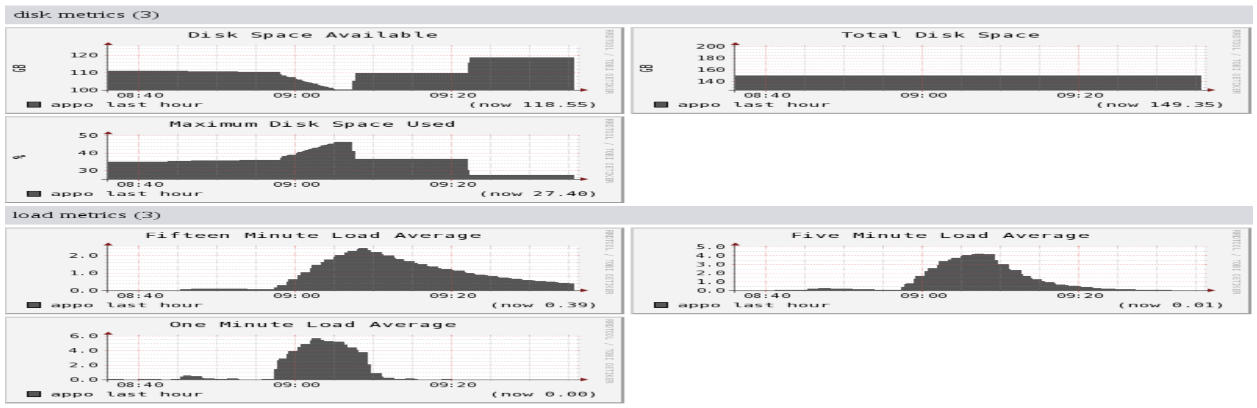


***Figure C-6**: Disk space error*

- The ACER class of machines have the least amount of hard disk and are thus the most likely to encounter such errors during a long task. To combat this, we gave the ACER nodes more space on another partition and also compressed the output of Map tasks written to disk.
    - `./clusterAdmin.sh - e “mkdir -p /tmp/hdfs/dfs/data”`
    - `./clusterAdmin.sh -e “sudo chown -R hdfs /tmp/hdfs”`
    - (hdfs-site.xml) dfs.datanode.data.dir -> /home/hdfs/dfs/data,/tmp/hdfs/dfs/data
    - (mapred-site.xml) mapreduce.output.fileoutputformat.compress -> true
    - `./clusterAdmin -h CLASS_ACER:CLASS_ACER`
    - `./clusterAdmin -r`
- When HDFS writes files, it does not by default take into account that one file partition may be larger than another listed. In our case, the partition that CentOS mounts on ‘/’ is only 50GB in size, smaller than the ‘/home’ partition. To prevent HDFS from writing until the partition is completely full, it was necessary to specify the amount of bytes to reserve on each partition so that log files and other incidental files do not exceed the available space on disk:
    - (hdfs-site.xml) dfs.datanode.du.reserved -> 10737418240
- In addition to increasing the default space on HDFS and compressing MapReduce output, we can specify that the temporary output of map jobs should be compressed when spilled to disk:
    - (mapred-site.xml) mapreduce.map.output.compress -> true
- If CheckSumException errors are encountered, the filesystem can be checked for corruption with ‘hadoop fsck /’
    - to delete corrupted files, ‘hadoop fsck / -delete’

With these changes, the cluster was capable of handling the sort task without failing multiple task attempts or failing completely. This configuration will serve as the initial BASIC configuration that will be further refined to increase performance.

Optimizing:

In many ways, configuring a YARN cluster is easier than the old MR1 clusters. Of primary concern is specifying a proper upper limit for the amount of memory and heap space that the individual nodes should make available to the ResourceManager and configuring other buffer sizes associated with the shuffle. As we've seen, the default memory per node is 8GB, a setting vastly too small for most industry clusters and yet vastly too large for the cluster used for this project. This misconfiguration can cause a large amount of spillage and thrashing, greatly retarding overall cluster performance.

By checking the ResourceManager Web UI (http:<resourceManager>:8088), information can be gained as to the amount of container launched during individual tasks. With just the BASIC configuration files, the Hadoop daemons launched 153 containers for the sort task during initial testing of the project cluster, with each one anticipating access to 1GB+ of memory resources. With few nodes in possession of such resources, a vast amount of spills would be necessary for the containers to complete their operations.

Configuring a Hadoop cluster can depend to a great extent upon the type of operations one is seeking to use it for. In general, the most useful portion of a Hadoop job to configure is the "shuffle" - the period during which Map output is sorted on the mappers and written to disk and reducers fetch and merge the outputted files as input into the Reduce phase.

As a general rule of thumb for deciding how many YARN containers to run on a specific machine, this approximation is often useful:

- # of containers = min (2*CORES, 1.8*DISKS, (Total available RAM) / MIN_CONTAINER_SIZE)

Before specifying the various configuration values we'll use and repeatedly benchmarking the cluster, it is necessary to understand what the various values actually affect during the execution of tasks (many are very poorly documented).

During benchmarking of the cluster, dozens of tests were performed pertaining to individual configuration values, with the most efficient noted and split into three rounds of implementation specified below for easier analysis:

1. YARN settings / Core-site / Hadoop-env
2. Mapred-site
3. Hdfs-site

Round 1 (yarn-site.xml):

- **yarn.scheduler.minimum-allocation-mb** -> this property specifies the smallest container that will be granted by the ApplicationMaster. Any requests for a lower amount will be bumped up to this amount. It makes sense for this value to correspond to the smallest amount of memory any class gives map tasks, to which it should give twice as much to reduce tasks. Since the LOW_END class will allow 1024, that value was also used here.
- **yarn.scheduler.maximum-allocation-mb** -> this property specifies the largest container that will be granted by the ApplicationMaster. Any requests for a greater amount of RAM will be capped at this value. It makes sense for this value to

correspond to the largest amount of memory any class gives reduce tasks, which was NEWO at 14336.

- **yarn.nodemanager.resource.cpu-cores** -> properties specifically tied to the local nodemanager are among the only node-specific YARN configurations that are settable on the cluster. This property corresponds to the number of CPU cores available to YARN on this node.
- **yarn.nodemanager.resource.memory-mb** -> the amount of memory made available to YARN on this specific node.
- **yarn.nodemanager.vmem-pmem-ratio** -> This property specifies the ratio of virtual to physical memory for containers. Raising it will allow weaker nodes to handle larger splits but at a cost of increased spills and decreased overall performance/throughput. Checks are performed (yarn.nodemanager.vmem-check-enabled, yarn.nodemanager.pmem-check-enabled) and the ApplicationMaster will remotely kill any containers that are in violation of this ratio.
- **yarn.app.mapreduce.am.resource.mb** -> Amount of memory resources to allocate to the ApplicationMaster. The default 1.5GB is small and will likely result in unused leftover (512mb) on many nodes in the cluster.
- **yarn.app.mapreduce.am.command-opts** -> Can specify JVM options such as maximum heap size for the ApplicationMaster.
- **yarn.app.mapreduce.am.resource.cpu-vcores** -> The number of virtual CPU cores allocated to the ApplicationMaster process.
- **yarn.dispatcher.exit-on-error** -> if set to true, YARN dispatcher daemons will System.exit in the event of errors. Useful to set this to true so that daemons don't have to be manually killed following crashes.

(yarn-env.sh):

- **YARN_HEAPSIZE** -> default heap space granted to YARN daemons (ResourceManagers, NodeManagers).

(core-site.xml):

- **ipc.server.listen.queue.size** -> the length of listen queues for servers accepting incoming client connections. This value should be set to correspond to the somaxxconn value set earlier (1024) in Appendix A. Otherwise the earlier RPC calls will get wiped by incoming connections if this value is not large enough.
- **io.file.buffer.size** -> size of buffer used for handling SequenceFiles. Should be a multiple of hardware page size (`getconf PAGESIZE`) and OS page size (`sudo blockdev --getbsz /dev/sda1`). If HDFS block size is kept at it's default value, this can be raised to match it (65536).
- **file.bytes-per-checksum** -> A 4-byte CRC-32 checksum will be computed after this many bytes when reading files from a raw local filesystem. To reduce storage overhead, increased from default 512 to 4096 (can still fit in OS page).
- **io.bytes.per.checksum** -> A 4-byte CRC-32 checksum will be computed after this many bytes of an IO stream. To reduce storage overhead, increased from default 512 to 4096 (can still fit in OS page).

- **dfs.bytes-per-checksum** -> A 4-byte CRC-32 checksum will be computed after this many bytes when read from HDFS. To reduce storage overhead, increased from default 512 to 4096 (can still fit in OS page).

(hadoop-env.sh):

- **HADOOP_HEAPSIZE** ->The standard amount of memory allocated to Hadoop daemons for heap space. These daemons generally launch child JVMs to run Map and Reduce tasks in. This value pertains only to the initial daemon tasks - namenodes, datanodes, resourcemanagers, nodemanagers, etc.
- **HADOOP_CLIENT_OPTS** -> Java runtime options when for Hadoop daemon clients (like fs, dfs, fsck, distcp, etc).
    - **"-XX+UseParallelOldGC**" -> explicitly set garbage collection to run in parallel mode for full collections (not just for scavenges of "nursery" heap area - automatically sets +UseParallelGC).
    - **"-XX:ParallelGCThreads=8"** -> use 8 threads for garbage collection. Generally the parallel algorithms offer a higher throughput for the sort of applications commonly run on Hadoop.
    - **"-server"** -> Oracle's Hotspot JVM has two different modes. Server mode has been specifically tuned to maximize operating speed for long-running applications. Works at the expense of start-up speed and memory footprint.
- **HADOOP_OPTS** -> The runtime options passed to the JVM executing the 'hadoop' executable. Passes any specific JVM flags to the specific node's JVM and then sends HADOOP_CLIENT_OPTS, too.

(capacity-scheduler.xml):

- **yarn.scheduler.capacity.maximum-am-resource-percent** -> the default value of 0.1 means that no more than 10% of cluster memory can be dedicated to running ApplicationMaster daemons. Since the project cluster consists of approximtely 65GB of RAM in its entirety and an ApplicationMaster daemon is configured to use at 2GB, this effectively caps the number of running jobs at 3. This is quite bad for throughput, as often (following configuration) jobs will bog down the resource-heavy nodes during the latter stages of Reduce, leaving many potential Mappers to lie unutilized.

Round 2 (mapred-site.xml):

- **mapreduce.[map/reduce].log.level** (mapred-site.xml) -> Hadoop generates an extremely large amount of logging information to assist in diagnosing MapReduce problems or performance bottlenecks. By default, all of this logging information is written immediately to the specified log output file, which can throw off the HD head and impede performace. Setting these levels to WARN (but not final) reduces the amount of writes that will be performed for logging while still allowing individual jobs to override this value if they're experiencing difficulties, etc. Hadoop's system-wide logging level can also be altered in it's log4j.properties file, but this requires more work to change by jobs at runtime.
- **mapreduce.task.userlog.limit.kb** (mapred-site.xml) -> By default, Hadoop will

cache (in memory) this amount of log data, waiting until job completion to write it to disk (thus not affecting the HD head during peak job I/O periods). The resultant log buffer is circular, meaning that any log data in excess of the value will erase prior data before being written to disk. Setting this configurable to a value such as 4096 will ensure that the logs will likely contain any log information pertinent to sudden crashes while not imposing too significant of a memory overhead on the weaker nodes in the cluster.

- **mapreduce.job.userlog.retain.hours** (mapred-site.xml) -> Number of hours to retain Mapreduce logs on the local filesystem.
- **mapreduce.map.memory.mb** -> Size of Map containers running on YARN. Is a hint to what the ResourceManager will request in resources from the NodeManagers in order to run a map task for this particular application. Should generally correspond to the yarn.scheduler.minimum-allocation-mb value so that all nodes can run map tasks. Note that this and all “mapreduce” settings are job-specific and cannot be custom configured for individual nodes in the cluster.
- **mapreduce.reduce.memory.mb** -> Size of Reduce containers running on YARN. Is a hint to what the ResourceManager will request in resources from the NodeManagers in order to run a reduce task. Generally the reduce phase is more resource intensive and can require merging large map outputs with significant spill costs. By setting this value to large enough to exclude the weakest nodes it is possible to further modify reduce heap space size and other buffers that will streamline this phase and use the larger memory resources of the strong nodes to their fullest potentials.
- **mapreduce.map.output.compress** -> This property determines whether the output from all map tasks should be compressed. While obviously adding additional computation, using compression when tasks are bound by network traffic (identified through Ganglia) can vastly increase performance.
- **mapreduce.map.output.compress.codec** -> Hadoop’s default DEFLATE compression algorithm uses a combination of the LZ77 algorithm and Huffman coding. Widely used in gzip, the DEFLATE algorithm is good at reducing size and performs moderately fast. One great issue with DEFLATE in a Hadoop environment however is its inability to split files at boundaries (such as HDFS blocksizes). This means that one node must store the whole file and that mappers processing that file will run entirely on that node (or other replicants). Alternatives to DEFLATE that allow for the splitting of files at arbitrary boundaries (generally the size of HDFS blocks) include BZip2’s Java implementation and LZO when pre-indexed.
    - Tests performed on the 20-node cluster indicate that network performance was not enough of a concern to justify using computationally expensive algorithms such as DEFLATE and BZip2, despite the latter’s splittable nature. The fastest implementation tested was Google’s SNAPPY compression library, which must be downloaded and manually built:
        - http://code.google.com/p/snappy/
        - ./configure; make; make install
        - cp /usr/local/lib/snappy* /usr/lib/hadoop/lib/
    - For larger clusters or jobs even more bandwidth-intensive than the

benchmarks performed, other compression algorithms may outperform SNAPPY. Use of Ganglia to identify bottlenecks is cruciall

- **mapreduce.[map/reduce].speculative** (hdfs-site.xml) -> this property defines whether the cluster will begin speculatively executing tasks (up to mapreduce.job.speculative.speculativecap) when it determines that a certain task or node is running slowly. By default, once a task has taken more than one standard deviation (34%) longer than the average, it is eligible to be launched speculatively on other nodes that have generally performed well in the past (above mapreduce.job.speculative.slownodethreshold). Given the vast hardware differences between the various nodes utilized throughout the project, tasks will often be executed speculatively when assigned to low-resource nodes. To maximize throughput, speculative execution during the Map phase was disabled while speculative execution was encouraged during the resource-intensive Reduce stage.
- **mapreduce.reduce.shuffle.input.buffer.percent** -> the amount of JVM heap space reserved in Reduce tasks for Mapper output. The main cause of Reducers being more resource-intensive than Mappers.
- **mapreduce.reduce.shuffle.merge.percent** -> when the buffer corresponding to mapreduce.reduce.shuffle.input.buffer.percent gets filled to this percentage, the accumulated outputs are merged and spilled onto disk.
- **mapreduce.reduce.input.buffer.percent** -> the percentage of memory reserved to hold map outputs during the reduce. If the reduce stage is light on memory requirements, this value can be cranked up from its default 0 to reduce spilling to the hard disk and having to pull back after the fetch. Before the Reduce() will start, map outputs must consume less than this percentage of memory.
- **mapreduce.map.sort.spill.percent** -> In a MapReduce program, the output key-value pairs are sorted in the Map phase prior to being shuffled into Reducers. This configuration specifies the maximum amount of the heap allocated for sorting that this sorting can occupy in memory before it will begin spilling to disk.
- **mapreduce.task.io.sort.mb** -> when map functions begin producing output, this output is not immediately written to disk. Each map task has a circular memory buffer that can temporarily store these results. This size of this buffer is determined by this property, which defaults to 100MB. Once this buffer fills up a certain percentage of the buffer (mapreduce.map.sort.spill.percent), it is spilled to disk.
    - In the results printed after a MR run, if “Map output records” is less than “Spilled Records”, the buffer was not large enough and spilling occurred.
- **mapreduce.map.java.opts** -> Can specify the maximum JVM heap-size for map tasks so as to avoid problems brushing up against container limits. It is set on a per-*JOB* basis. The default value is -Xmx1000m. This value corresponds to the absolute cap on *physical* heap memory used by map tasks.
- **mapreduce.map.java.opts** -> Can specify the maximum JVM heap-size for reduce tasks so as to avoid problems brushing up against container limits. It is set on a per-*JOB* basis. The default value is -Xmx1000m. This value corresponds to the absolute cap on physical heap memory used by reduce tasks.
- **mapreduce.tasktracker.reduce.tasks.maximum** -> a “hint” to the MapReduce architecture to run a limit reduce tasks to this setting. Is not a hard cap, but may be

helpful to set low on weak nodes due to the high latency costs of assigning intense reduce tasks to weak nodes.
- **mapreduce.tasktracker.map.tasks.maximum** -> the default is 8, more than enough for the project cluster.

Round 3 (hdfs-site.xml):

- **dfs.blocksize** -> asf
- **fs.file.impl** -> Long Hadoop jobs running on non-ECC memory (`sudo dmidecode -t memory` to check) are likely to experience many checksum errors despite the underlying filesystem functioning as it should. To disable checksums, switch from the default org.apache.hadoop.fs.LocalFileSystem to org.apache.hadoop.fs.RawLocalFileSystem. Must be set by executing job.
- **dfs.datanode.handler.count** -> number of RPC server threads to run on datanodes. Solves 'NotReplicatedYet' errors.
- **dfs.namenode.handler.count** -> number of RPC server threads to run on the namenode. Solves 'NotReplicatedYet' errors.
- **dfs.replication** -> the replication factor for files placed in HDFS. Increasing this value will increase network traffic but also can increase performance on pre-written data due to increased locality/availability of local containers.
- **dfs.client.read.shortcircuit** -> When executing anything on Hadoop HDFS, reads normally must go through the local datanode. When a client wants to read a file, they contact the datanode that possesses it and that datanode subsequently reads the file from disk and sends it over a TCP socket. Short-circuit reads bypass the datanode entirely and allow the client to read the file directly from the local filesystem. This direct access can provide large performance benefits for certain access patterns.
    - Note: Native Libraries must be present and property configured for short-circuit reads to work properly. It is simple to compile these from the Hadoop source package:
        - install a C compiler (e.g. GNU C compiler)
        - install the GNU autotools chain: autoconf, automake, libtool
        - install the latest stable zlib-development package
        - once the source package is downloaded and extracted, invoke Maven from the main pom.xml: `mvn package -Pdist,native -Dskiptests -Dtar`
        - resulting native libraries should be found in hadoop-dist/target/hadoop-2.3.0/lib/native.
        - copy native libraries into $HADOOP_HOME/lib/native
        - in core-default.xml, set 'io.native.lib.available' to 'true' & 'mapreduce.admin.user.env' to 'LD_LIBRARY_PATH=$HADOOP_COMMON_HOME/lib/native'
- **dfs.domain.socket.path** -> short-circuit reads used standard UNIX domain sockets. This path must be accessible to the datanode process but secure to all other users (except hdfs and root).
    - `sudo ./clusterAdmin.sh -e "touch`

```
/var/lib/hadoop-hdfs/dn_socket;chown hadoop
/var/lib/hadoop-hdfs/dn_socket;chmod -R 770
/home/hdfs/dfs/data;chmod -R 770
/tmp/hdfs/dfs/data;usermod -a -G hdfs hadoop; chgrp
-R hdfs /home/hdfs/dfs/data; chgrp -R hdfs
/tmp/hdfs/dfs/data"
```

  - `sudo ssh aho -t "touch /var/lib/hadoop-hdfs/dn_socket;chown hadoop /var/lib/hadoop-hdfs/dn_socket;chmod -R 770 /home/hdfs/dfs/data;chmod -R 770 /tmp/hdfs/dfs/data;usermod -a -G hdfs hadoop; chgrp -R hdfs /home/hdfs/dfs/data; chgrp -R hdfs /tmp/hdfs/dfs/data"`
  - normally users have to logout and back in for their group assignments to update. Instead,
    - `./clusterAdmin -e “exec su -l hadoop”`
- **dfs.block.local-path-access.user** -> user allowed to use the shortcut domain socket for reading. Should correspond to the hadoop user (and hbase, oozie, yarn if running). To add the ability for more users to perform short-circuit reads, the dfs.datanode.data.dir.perm configuration should be edited to 740 and the intended user added to hdfs supergroup.
- **dfs.datanode.data.dir.perm** -> the permissions expected on the HDF data directory. If actual permissions differ from this, the system will automatically reset them to this value.
- **dfs.datanode.hdfs-blocks-metadata.enabled** -> enabled support for the new (experimental) DistributedFileSystem#getFileVBlockStorageLocationsAPI. Datanodes will not keep block and disk information cached locally to avoid constant queries to the namenode.
- **dfs.client.file-block-storage-locations.timeout** -> timeout cap in milliseconds for parallel RPC calls made to the DFS. When stack traces begin showing errors in calls to DistributedFileSystem#getFileBlockStorageLocations() - which happens a lot - it’s good to increase this value.
- **dfs.client.use.legacy.blockreader.local** -> since short circuit reads depend on UNIX-type domain sockets (IPC socket similar to a pipe), Windows users don’t have access to this functionality. Running on CentOS, we want to make sure this is disabled so that the more efficient IPC blockreader is utilized.
- **dfs.permissions.superusergroup** -> the name of the groups of HDFS superusers. The ‘hadoop’ and ‘hdfs’ user should be a member.
  - `sudo groupadd supergroup`
  - `sudo usermod -a -G supergroup hdfs`
  - `sudo usermod -a -G supergroup hadoop`
  - `exec su -l hdfs`
  - `exec su -l hadoop`
- **fs.permissions.umask-mode** -> the umask value used for files written from HDFS onto the local filesystem. Once short-circuit reads are enabled, the ‘hadoop’ user

must have access to these files as well, which changing this value from it's default 022 to 002 will accomplish (leave off sticky bit).

- **dfs.client.read.shortcircuit.skip.checksum** -> disable checksum calculations when files are read through short-circuit reads. Will speed performance at the cost of increased risk of data loss; with non-ECC memory however checksum issues will likely be a constant annoyance otherwise.
- **dfs.permissions.enabled** -> turns off all permission checks for the HDFS filesystem. Useful for ascertaining the nature of some problems, specifically whether permission/block issues originate with the local filesystem or HDFS. Should not be left disabled.

Round 3 system tweaks:

- **Inode size** -> Inodes are the filesystem data structure units that determine how files and directories are stored. Filesystems with smaller inode sizes will be able to store more files, but filesystems with large inode sizes will experience higher I/O throughput. To raise the limit:
    - `sudo tune2fs -l /dev/sdb1 | grep Inode`
    - `sudo mke2fs -j -I 4096 /dev/sdb1`
    - `sudo tune2fs -l /dev/sdb1 | grep Inode`
- **Read-ahead buffer** -> When Linux reads a file from disk, it can optionally keep on reading past the requested data and store this data in a buffer, anticipating that the program will subsequently need this data as well. By default (can be checked with `sudo blockdev --report`) the kernel allocated 128K (in 512-byte sectors) of memory for this buffer. Since Hadoop is very likely to sequentially read vastly larger amounts of contiguous disk data, increasing this buffer can be helpful:
    - `sudo blockdev --setra 32768 /dev/sdb`
    - to make this readahead setting permanent, add the above command to the /etc/rc.local file.

To assist in the discovery and configuration of Hadoop configurables, the config program was added to TextProject. The pertinent aspects of the code are the loading of default Hadoop configurables, loading the custom .xml files that over-ride these values and then displaying any deprecated keys (common occurrence between successive distributions):

```
Configuration.addDefaultResource("hdfs-default.xml");
Configuration.addDefaultResource("mapred-default.xml");
Configuration.addDefaultResource("yarn-default.xml");
Configuration.addDefaultResource("core-default.xml");
Configuration conf = getConf();
conf.addResource(new Path("file:///etc/hadoop/conf/hdfs-site.xml"));
conf.addResource(new Path("file:///etc/hadoop/conf/mapred-site.xml"));
conf.addResource(new Path("file:///etc/hadoop/conf/yarn-site.xml"));
conf.addResource(new Path("file:///etc/hadoop/conf/core-site.xml"));
for (Entry<String, String> entry : conf)
     System.out.printf("%s=%s\n", entry.getKey(), entry.getValue());
System.out.println("*** Deprecated keys: ***");
Configuration.dumpDeprecatedKeys();
```

***Figure C-7**: Configuration printing*

Once finished, the clusterAdmin script can be very helpful in spreading the new configurations throughout the cluster.

- `./clusterAdmin.sh -h ACER_CLASS:ACER_CLASS`
- `./clusterAdmin.sh -h LOW_END:LOW_END`
- `./clusterAdmin.sh -h MID_RANGE:MID_RANGE`
- `./clusterAdmin.sh -h NEWO:NEWO`
- `./clusterAdmin.sh -h MINO:MINO`
- `./clusterAdmin.sh -h APPO:APPO`

Or to spread new Hadoop configurations to all classes and nodes:

- `./clusterAdmin.sh -a`

And then restart the HDFS/MapReduce slave daemons:

- `sudo ./clusterAdmin.sh -r`

Results (in minutes):

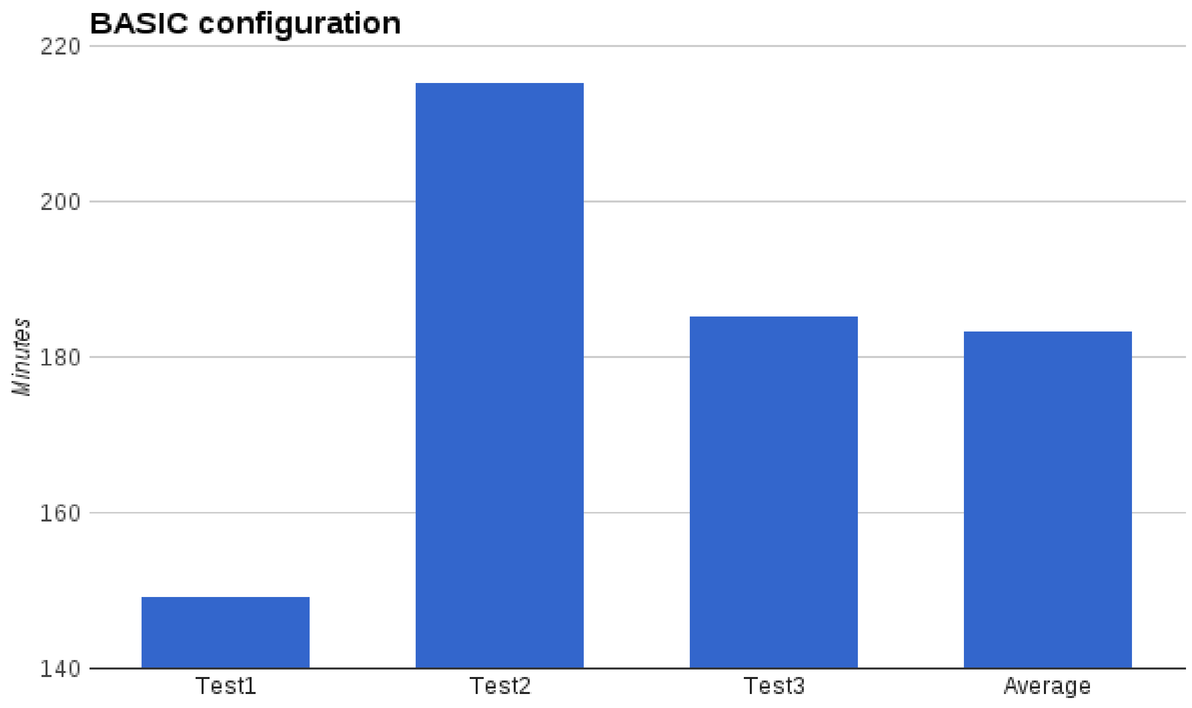


***Figure C-8**: Initial sort performance*

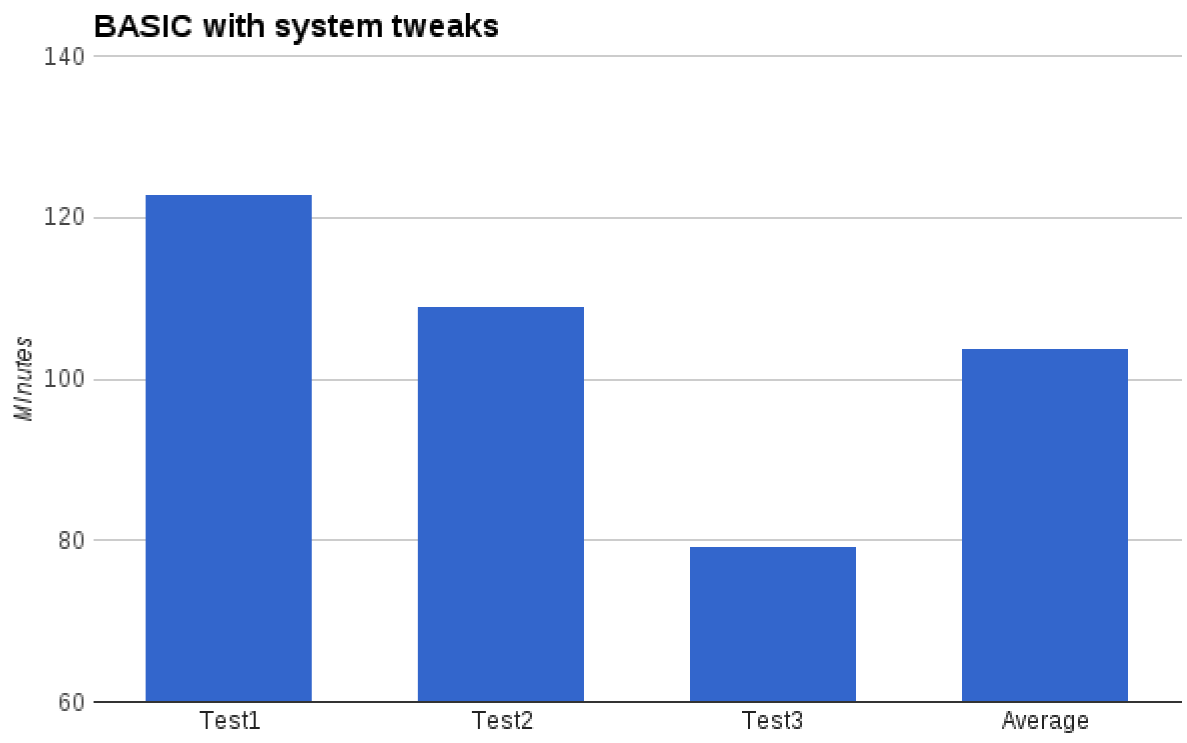


***Figure C-9**: Sort performance after system tweaks*

Yarn settings:

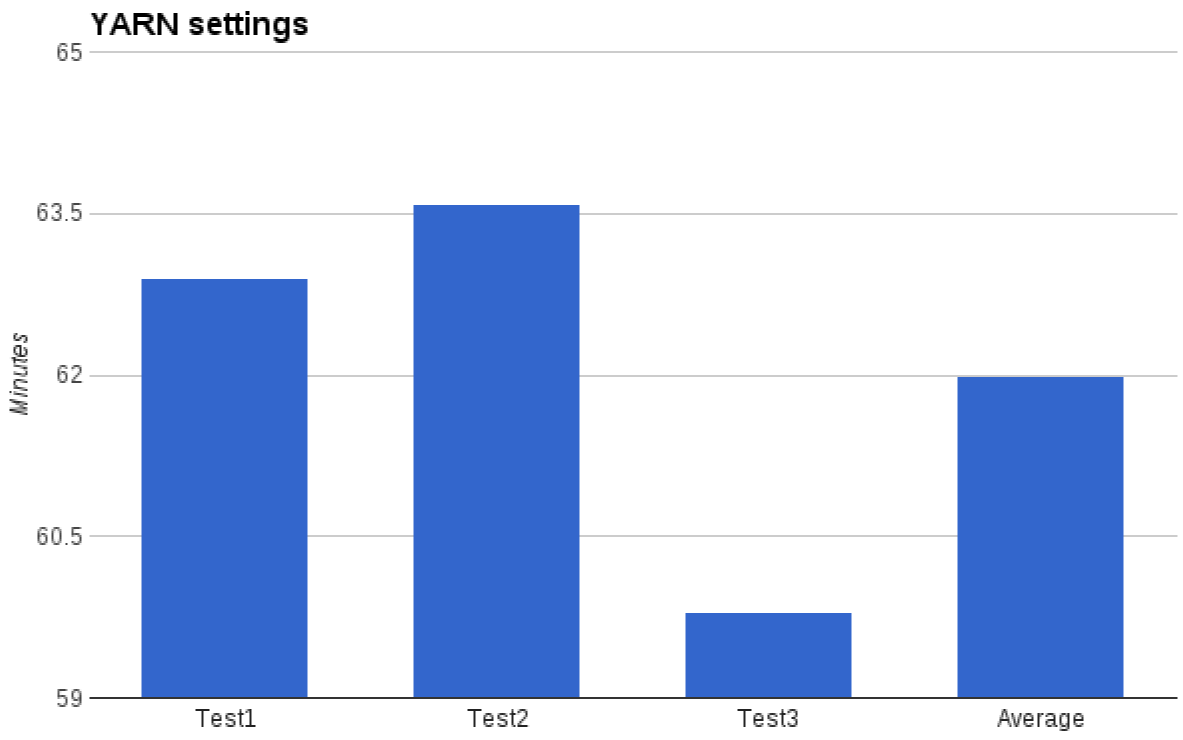


***Figure C-10**: Performance after YARN configuration*

Mapred:

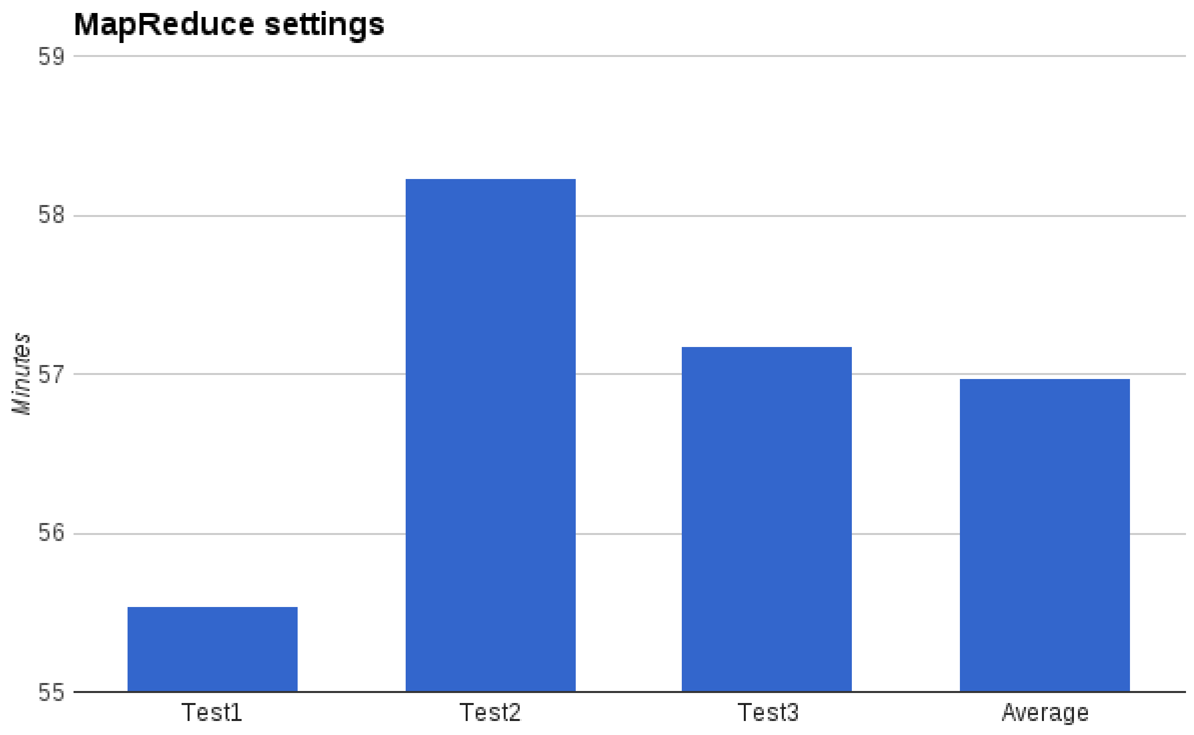


Figure C-11*: Sort performance after MapReduce configurations*

HDFS-Settings:

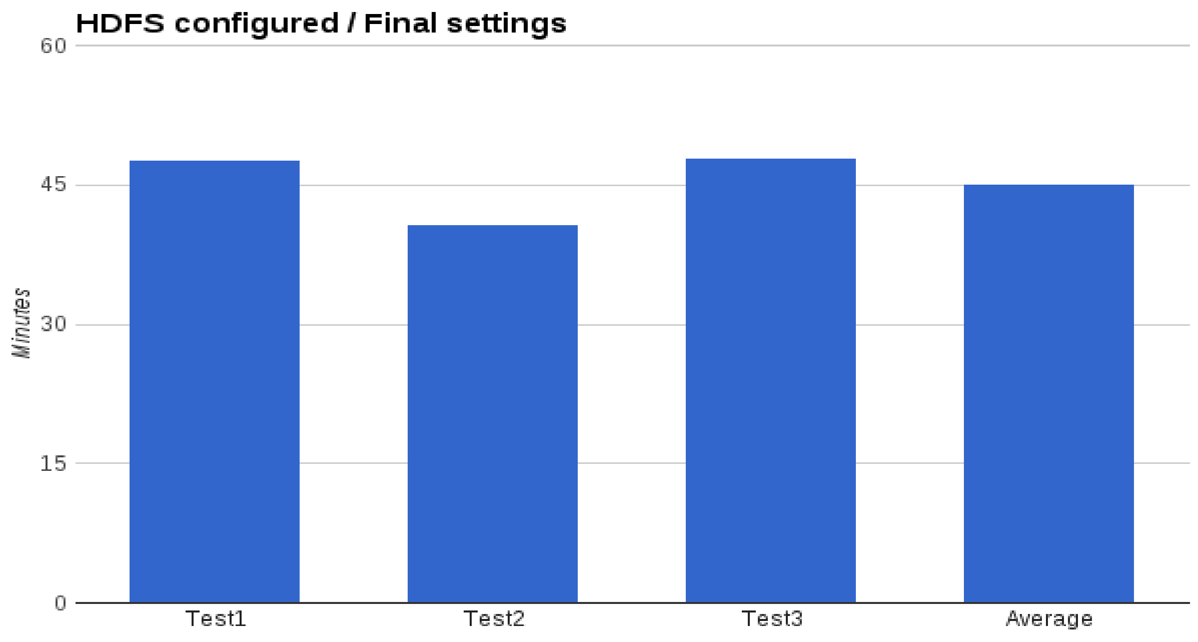


Figure C-12: Sort performance after HDFS configuration

The final cluster configurations resulted in an over 400% speed-up compared to the BASIC setup. The final configuration of the cluster: